

Dezun Dong · Xiaoli Gong ·
Cunlu Li · Dongsheng Li ·
Junjie Wu (Eds.)

Communications in Computer and Information Science

1256

Advanced Computer Architecture

13th Conference, ACA 2020
Kunming, China, August 13–15, 2020
Proceedings

 Springer



Communications in Computer and Information Science

1256

Commenced Publication in 2007

Founding and Former Series Editors:

Simone Diniz Junqueira Barbosa, Phoebe Chen, Alfredo Cuzzocrea,
Xiaoyong Du, Orhun Kara, Ting Liu, Krishna M. Sivalingam,
Dominik Ślęzak, Takashi Washio, Xiaokang Yang, and Junsong Yuan

Editorial Board Members

Joaquim Filipe 


Polytechnic Institute of Setúbal, Setúbal, Portugal

Ashish Ghosh

Indian Statistical Institute, Kolkata, India

Igor Kotenko 

*St. Petersburg Institute for Informatics and Automation of the Russian
Academy of Sciences, St. Petersburg, Russia*

Raquel Oliveira Prates 

Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil

Lizhu Zhou

Tsinghua University, Beijing, China

More information about this series at <http://www.springer.com/series/7899>

Dezun Dong · Xiaoli Gong · Cunlu Li ·
Dongsheng Li · Junjie Wu (Eds.)

Advanced Computer Architecture

13th Conference, ACA 2020
Kunming, China, August 13–15, 2020
Proceedings

Editors

Dezun Dong
National University of Defense Technology
Changsha, China

Xiaoli Gong
Nankai University
Tianjin, China

Cunlu Li
National University of Defense Technology
Changsha, China

Dongsheng Li
National University of Defense Technology
Changsha, China

Junjie Wu
National University of Defense Technology
Changsha, China

ISSN 1865-0929

ISSN 1865-0937 (electronic)

Communications in Computer and Information Science

ISBN 978-981-15-8134-2

ISBN 978-981-15-8135-9 (eBook)

<https://doi.org/10.1007/978-981-15-8135-9>

© Springer Nature Singapore Pte Ltd. 2020

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd. The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

It was our pleasure to welcome all of you to the technical program of the 13th Conference on Advanced Computer Architecture (ACA 2020). ACA is the premier forum for the presentation of research results in computer architecture in China. ACA is mostly held every two years and has evolved nearly three decades. ACA 2020 is hosted by the China Computer Federation (CCF) and co-organized by the CCF Technical Committee on Computer Architecture and National University of Defense Technology, China. ACA 2020 was held online, which will be remembered as the first time since its inception, due to the COVID-19 outbreak. The theme of ACA 2020 was “Computer Architecture Entering the New Golden Age.” The conference focused on state-of-the-art computer architecture issues and featured a series of exciting and rich activities, including invited keynotes, technical reports, and professional forums.

We solicited papers on all aspects of research, development, and application of computer architecture. We received 149 paper registrations and 105 complete submissions. Each submission was reviewed by three Program Committee (PC) members on average. There was also an online discussion stage to guarantee that consensus was reached for each submission. Finally, the PC decided to accept 64 submissions, including 24 papers in English and 40 papers in Chinese. 7 of the 64 accepted papers were conditionally accepted. Authors of all the accepted papers were asked to submit a revised version based on the review comments. Each conditionally accepted paper was shepherded by one assigned PC member, and further revised at least one round before the camera-ready deadline.

We would like to thank all the colleagues who submitted papers and congratulate those whose papers were accepted. We would like to thank all PC members and additional reviewers for their contribution to the program. Their names are listed in the subsequent pages. The PC members did an excellent job in returning high-quality reviews in time and engaging in a constructive online discussion. Without their efforts, this program would have not been possible. We would like to express our deepest gratitude to the publication chairs, Xiaoli Gong and Cunlu Li, who made a great effort to communicate frequent reminders to authors and give feedback to the CCF conference review system with new demands. Our thanks also go to Springer for their assistance in putting the proceedings together. We would like to thank all the other people involved in the organization of ACA 2020: general chairs, Ninghui Sun and Depei Qian; executive chair, Dongsheng Li; Steering Committee chairs, Yong Dou and Chenggang Wu; publicity chairs, Li Shen and Chao Wang; workshops chair, Zichen Xu; and website chair, Zhen Huang.

July 2020

Dezun Dong

Organization

General Chairs

Ninghui Sun Institute of Computing Technology, Chinese Academy
 of Sciences, China
Depei Qian Sun Yat-sen University, China

Steering Committee Chairs

Yong Dou National University of Defense Technology, China
Chenggang Wu Institute of Computing Technology, Chinese Academy
 of Sciences, China

Steering Committee

Zhenzhou Ji Harbin Institute of Technology, China
Dongsheng Wang Tsinghua University, China
Xingwei Wang Northeastern University, China
Gongxuan Zhang Nanjing University of Science and Technology, China
Junjie Wu National University of Defense Technology, China
Tao Li Nankai University, China
Dongsheng Li National University of Defense Technology, China
Chao Li Shanghai Jiao Tong University, China
Li Shen National University of Defense Technology, China

Executive Chair

Dongsheng Li National University of Defense Technology, China

Local Chair

Lijun Yun Yunnan Normal University, China

Program Chair

Dezun Dong National University of Defense Technology, China

Publicity Chairs

Li Shen National University of Defense Technology, China
Chao Wang University of Science and Technology of China, China

Workshops Chair

Zichen Xu Nanchang University, China

Publication Chairs

Xiaoli Gong Nankai University, China
Cunlu Li National University of Defense Technology, China

Web Chair

Zhen Huang National University of Defense Technology, China

Program Committee

Hong An University of Science and Technology of China, China
Qiang Cao Huazhong University of Science and Technology,
China

Lizhong Chen Oregon State University, USA
Quan Chen Shanghai Jiao Tong University, China
Yunji Chen Institute of Computing Technology, Chinese Academy
of Sciences, China

Chen Ding University of Rochester, USA
Zhenman Fang Simon Fraser University, Canada
Xiaobing Feng Institute of Computing Technology, Chinese Academy
of Sciences, China

Xiaoli Gong Nankai University, China
Binzhang Fu Huawei, China
Bingsheng He National University of Singapore, Singapore
Yang Hu The University of Texas at Dallas, USA
Yu Hua Huazhong University of Science and Technology,
China

Weixing Ji Beijing Institute of Technology, China
Jingwen Leng Shanghai Jiao Tong University, China
Chao Li Shanghai Jiao Tong University, China
Dongsheng Li National University of Defense Technology, China
Tao Li Nankai University, China
Yun Liang Peking University, China
Chi Lin Dalian University of Technology, China
Duo Liu Chongqing University, China
Haikun Liu Huazhong University of Science and Technology,
China

Xu Liu College of William and Mary, USA
Xiaoyi Lu Ohio State University, USA
Jiahua Lu Xilinx, China

Songwen Pei	University of Shanghai for Science and Technology, China
Pengju Ren	Xian Jiaotong University, China
Li Shen	National University of Defense Technology, China
Shuaiwen Leon Song	The University of Sydney, Australia
Tian Song	Beijing Institute of Technology, China
Guangyu Sun	Peking University, China
Chao Wang	University of Science and Technology of China, China
Lei Wang	National University of Defense Technology, China
Bo Wu	Colorado School of Mines, USA
Chenggang Wu	Institute of Computing Technology, Chinese Academy of Sciences, China
Yuan Xie	University of California, Santa Barbara, USA
Zichen Xu	Nanchang University, China
Hailong Yang	Beihang University, China
Xiaochun Ye	Institute of Computing Technology, Chinese Academy of Sciences, China
Zhibin Yu	Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
Jidong Zhai	Tsinghua University, China
Weihua Zhang	Fudan University, China

Contents

Interconnection Network, Router and Network Interface Architecture

SDNVD-SCADA: A Formalized Vulnerability Detection Platform in SDN-Enabled SCADA System	3
<i>Jinjing Zhao, Ling Pang, and Bai Lin</i>	
Optimal Implementation of In-Band Network Management for High-Radix Switches	16
<i>Jijun Cao, Mingche Lai, Xingyun Qi, Yi Dai, and Zhengbin Pang</i>	
A 32 Gb/s Low Power Little Area Re-timer with PI Based CDR in 65 nm CMOS Technology	31
<i>Zhengbin Pang, Fangxu Lv, Weiping Tang, Mingche Lai, Kaile Guo, Yuxuan Wu, Tao Liu, Miaomiao Wu, and Dechao Lu</i>	
DBM: A Dimension-Bubble-Based Multicast Routing Algorithm for 2D Mesh Network-on-Chips	43
<i>Canwen Xiao, Hui Lou, Cunlu Li, and Kang Jin</i>	
MPLG: A Multi-mode Physical Layer Error Generator for Link Layer Fault Tolerance Test	56
<i>Xingyun Qi, Pingjing Lu, Jijun Cao, Yi Dai, Mingche Lai, and Junsheng Chang</i>	

Accelerator-Based, Application-Specific and Reconfigurable Architecture

GNN-PIM: A Processing-in-Memory Architecture for Graph Neural Networks	73
<i>Zhao Wang, Yijin Guan, Guangyu Sun, Dimin Niu, Yuhao Wang, Hongzhong Zheng, and Yinhe Han</i>	
A Software-Hardware Co-exploration Framework for Optimizing Communication in Neuromorphic Processor	87
<i>Shiying Wang, Lei Wang, Ziyang Kang, Lianhua Qu, Shiming Li, and Jinshu Su</i>	
A CNN Hardware Accelerator in FPGA for Stacked Hourglass Network	101
<i>Dongbao Liang, Jiale Xiao, Yangbin Yu, and Tao Su</i>	

PRBN: A Pipelined Implementation of RBN for CNN Training 117
*Zhijie Yang, Lei Wang, Xiangyu Zhang, Dong Ding, Chuan Xie,
and Li Luo*

Processor, Memory, and Storage Systems Architecture

Network-on-Chip Aware Task Mappings 135
Xiaole Sun, Yong Dong, Juan Chen, and Zheng Wang

Dissecting the Phytium 2000+ Memory Hierarchy via
Microbenchmarking. 150
Wanrong Gao, Jianbin Fang, Chuanfu Xu, and Chun Huang

TSU: A Two-Stage Update Approach for Persistent Skiplist 163
Shucheng Wang and Qiang Cao

NV-BSP: A Burst I/O Storage Pool Based on NVMe SSDs 178
Qiong Li, Dengping Wei, Wenqiang Gao, and Xuchao Xie

Pin-Tool Based Execution Backtracking. 192
Shuangjian Wei, Weixing Ji, Qiurui Chen, and Yizhuo Wang

Model, Simulation and Evaluation of Architecture

Directory Controller Verification Based on Genetic Algorithm 209
Li Luo, Li Zhou, Hailiang Zhou, Quanyou Feng, and Guoteng Pan

Prediction and Analysis Model of Telecom Customer Churn Based
on Missing Data 221
Rui Zeng, Lingyun Yuan, Zhixia Ye, and Jinyan Cai

How to Evaluate Various Commonly Used Program Classification
Methods? 233
Xinxin Qi, Yuan Yuan, Juan Chen, and Yong Dong

A Performance Evaluation Method for Machine Learning Cloud 249
Yue Zhu, Shazhou Yang, Yongheng Liu, Longfei Zhao, and ZhiPeng Fu

Parallelization and Optimization of Large-Scale CFD Simulations
on Sunway TaihuLight System 260
Hao Yue, Liang Deng, Dehong Meng, Yuntao Wang, and Yan Sun

New Trends of Technologies and Applications

Liquid State Machine Applications Mapping for NoC-Based
Neuromorphic Platforms 277
Shiming Li, Lei Wang, Shiyong Wang, and Weixia Xu

Compiler Optimizing for Power Efficiency of On-Chip Memory 290
Wei Wu, Qi Zhu, Fei Wang, Rong-Fen Lin, and Feng-Bin Qi

Structural Patch Decomposition Fusion for Single Image Dehazing 304
Yin Gao, Hongyun Li, Yijing Su, and Jun Li

Historic and Clustering Based QoS Aggregation for Composite Services 315
Zhang Lu and Ye Heng Zhou

**A High-Performance with Low-Resource Utility FPGA Implementation
of Variable Size HEVC 2D-DCT Transform. 325**
Ying Zhang, Gen Li, and Lei Wang

Author Index 335

Interconnection Network, Router and Network Interface Architecture



SDNVD-SCADA: A Formalized Vulnerability Detection Platform in SDN-Enabled SCADA System

Jinjing Zhao¹(✉), Ling Pang¹, and Bai Lin²

¹ National Key Laboratory of Science and Technology on Information System Security, Beijing, China

zhjj0420@126.com, stissl@163.com

² Beijing Institute of System Engineering, Beijing, China

linbaipeking@126.com

Abstract. After the Stunex event in 2010, the security problems of SCADA reveal to the public, which abstract more and more researchers to design new security firms to address the security problems of SCADA. Especially, after the software defined network (SDN) arose, it has become a beneficial attempt to improve the SCADA security. In this paper, a formalized vulnerability detection platform named SDNVD-SCADA is presented based on the SDN technology, which can be used to find the most familiar vulnerabilities in SCADA design, implementation, deployment and action processes. A general security mechanism description language and a SCADA vulnerability pattern database are embedded in SDNVD-SCADA to achieve the ambition of automatic vulnerability detection.

Keywords: SCADA · Software defined network · Vulnerability detection

1 Introduction

Supervisory control and data acquisition (SCADA) networks perform critical tasks and provide essential services within critical infrastructure, which be considered to be the backbone of any country. Critical infrastructure, and in particular control systems, require protection from a variety of cyber threats that could compromise their ordinary operation. The impairment of SCADA networks could cause interruption of critical services, process redirection, or manipulation of operational data that could have serious consequences for the population.

In terms of security, SCADA systems have many problems that might cause attacks or security events [1], e.g.:

1) Insecure design and implementation:

- **No hardware authentication:** which makes it easier to connect non-authorized computers to the system.

- **No access credentials:** several systems do not use access credentials, which means their security really only on the belief that no one (non-authorized) will get virtual or physical access to them.
 - **No individual access credentials:** some systems only allow the setting of general passwords that quickly become known by too many people.
 - **Uncontrolled access:** access limitations in control software are often not used.
 - **Anonymous access allowed:** services like Telnet and FTP often allow for anonymous login.
- 2) **No system patching:** on the last years several efforts were made to improve the policy on patch management and from these efforts resulted some standards [2]. Nevertheless, once systems go into production, they will likely never be patched, due to the impossibility of having downtime on the production line, the fear that systems may become unstable, have limitations, lack support/updates by the vendor, among others.
 - 3) **External network connections:** nowadays almost all the SCADA systems are, directly or indirectly, connected to external networks like internet, however it is often believed they are completely isolated from the outside world. This means that numerous connections are uncontrolled.

Many security firms have started designing solutions to address security problems of SCADA systems from different aspects. But there still has no efficient way to find its vulnerabilities automatically and correctly, because getting the security mechanisms of each entity and the whole network running state on time and on purpose are very hard to implement. But they are the fundamental factors of SCADA system security analysis.

SDN is an architecture that decouples forwarding functions (data plane) and network control (control plane), with the aim of introducing direct programmability into the network, to applications and policy engines alike. In recent years, many creative works have been done to deploy SDN in SCADA system [2, 6, 13]. The SCADA masters can also be operated as the SDN controllers or connect to a openflow switch just like the PLC. The network architecture can be designed as Fig. 1. With the help of SDN technology, the controllers can get all the entity states, including the security mechanisms and the traffic information of interactions between entities inside or outside the SCADA system.

In this paper, a formalized vulnerability detection platform named SDNVD-SCADA is presented based on the SDN technology, which can be used to find the most familiar vulnerabilities in SCADA design, implementation, deployment and action processes. In short, our paper makes the following contributions:

- 1) We propose the first SDN-enabled vulnerability detection platform in SCADA. We also implement a fast prototype on open source SDN controller Floodlight v1.0.
- 2) We investigate the familiar vulnerability patterns in SCADA and build the SCADA vulnerability database. The relations between these vulnerabilities and possible attacks are also be analyzed.
- 3) A SCADA security mechanism description language is designed in SDNVD-SCADA platform, which can be used to describe the security mechanisms of different

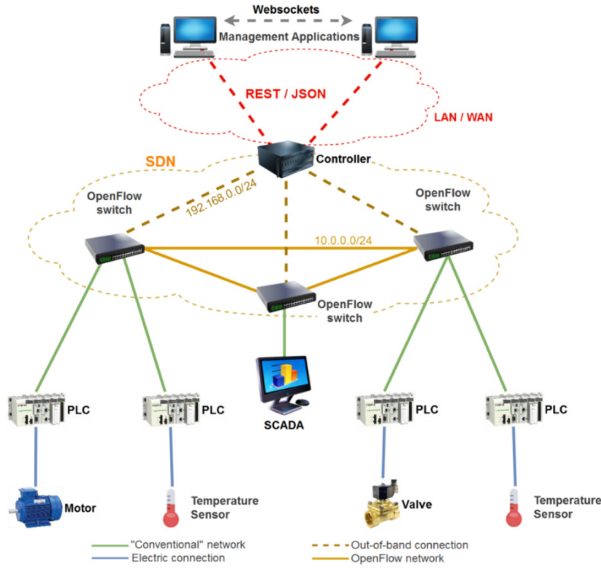


Fig. 1. A case of SDN deployed in SCADA system. The SCADA masters can also be operated as the SDN controllers or connect to a openflow switch just like the PLC.

entities based on their interaction modes. This makes the standardized and automated vulnerabilities analysis possible.

The remainder of this paper is organized as follows. The related works on SCADA security using SDN technology are listed in Sect. 2. The vulnerability detection method is presented in Sect. 3 with the introduction of SDNVD-SCADA architecture. The SDNVD-SCADA implementation is described in Sect. 4, including the SCADA security mechanism description language, SCADA vulnerability pattern database and SCADA vulnerability detector. An example is shown in Sect. 5 to illuminate how to use SDNVD-SCADA. Finally, we conclude in Sect. 6 with a brief summary and discussion.

2 Related Works

Using the SDN technology to improve the security of SCADA and ICS (Industrial Control System) is a new and creative attempt in recent years [2–14].

A approach is suggested by Machii et al. [4] as a way to minimize the attack surface by using SDN to dynamically segregate fixed functional groups within the ICS. This strategy reduces the time and spatial exposure to attacks (effectively creating a moving target) and also provides the means to isolate compromised devices.

Also related to dynamic configuration techniques, Chavez et al. [5] present a security solution based on network randomization, which also encompasses an IDS with near real-time reaction capabilities. This network randomization approach assigns new addresses to network devices in a periodic basis or by request, in order to protect them against attacks that rely on knowledge about the ICS topology (such as static device addresses).

Silva et al. [6] also describe a dynamic technique that makes use of SDN to prevent eavesdropping on SCADA networks. The intended goal is to deter attackers from collecting sequential data, which is essential for breaking encryption, identifying patterns, and retrieving useful information from the payload.

Genge et al. [7] propose two distinct SDN-based techniques to mitigate and block ICS cyber-attacks. The first technique, designed for single-domain networks, attempts to mitigate DoS attacks by rerouting traffic, using information from the SDN controller.

Song, Shin, and Choy [8] suggested using honeynets (networks set up with several honeypot devices) together with SDN technologies to detect scouting procedures and collect profiling information about attackers. Despite being a generic proposal, this solution can be easily ported to most ICS infrastructures.

Adrichem et al. [9] present a SDN network failover solution that should reduce the recovery time in multiple topologies. Being the speed of failure detection the main key for improving the recovery time, a short discussion on different failure detecting systems is presented and the best choice is presented in a detailed mode.

In [10], N. Dorsch et al. also describe their algorithms and different approaches based on SDN regarding: the efficiency improvement of network recovery time in case of link failure; the real-time processing of messages through the implementation of QoS mechanisms, based on the flexibility of SDN networks.

Rui Miguel's master degree dissertation [12] addresses the absence of proper management and security policies problems to improve SCADA ICS manageability, availability and security based on synergies between SDN and ICS domain.

In thesis [14], an SDN-assisted middleware is designed and implemented with open source platforms Open Network Operating System (ONOS) and Mininet, which not only enables real-time information exchange between two SCADA control centers but also supports multiple-to-multiple communications simultaneously.

3 SDNVD-SCADA Architecture

In the SDN-enabled SCADA system, SDNVD-SCADA can be installed on the SDN controller. Every important entity in SCADA which may influence the system security, should describe its security mechanisms and report them to the SDN controller by open-flow switch. Based on the common description of security mechanisms that the SCADA entities submitted, SDNVD-SCADA dissects them from the global view and utilizes the FSM to discover the vulnerabilities caused by the absence of some security mechanisms. The inputs of SDNVD-SCADA are the standard description of entity security mechanisms and the network running states, while the outputs are variety of potential vulnerabilities. The design of SDNVD-SCADA architecture is shown in Fig. 2.

In order to describe the security mechanisms of entities formally, SDNVD-SCADA proposes a uniform description language, which is a high level abstraction of entity security mechanism. The entity security mechanism is divided into three levels: system-level, Internet-level and operation level. Based on the study of various SCADA vulnerability cases and data, SCADA vulnerability pattern database defines the vulnerability patterns as the missing the confidentiality, integrity or availability of the critical resources in

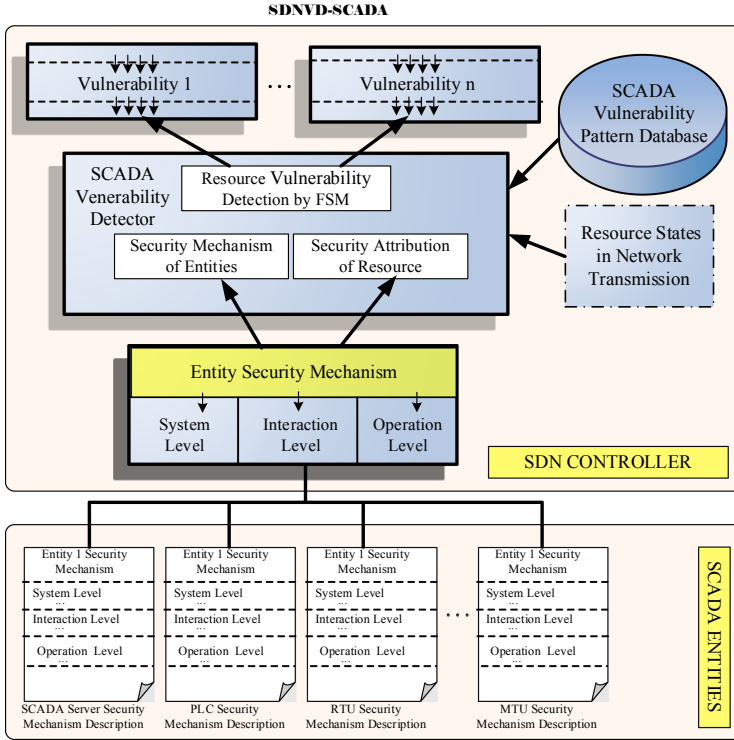


Fig. 2. The design of SDNVD-SCADA architecture, which can be separated into the controller part and entities part.

SCADA. The key characteristics of vulnerability are represented with the vulnerability name, triggered position and possible attack. In addition, SDNVD-SCADA sets the security sensitive resources as the analysis objects, extracts their security attributes from the SCADA security mechanisms of related entities, and build their finite-state machines (FSM) on their security states in the network transmission process. If the state of resource matches some items in the SCADA vulnerability pattern database, and the resource is sensitive to this vulnerability patterns, SDNVD-SCADA will deduce that the behavior of SCADA entities may cause a vulnerability.

The time cost of the process is composed of two parts. The one is the time of resources information generation, which is $O(|Entity|)$ and the $|Entity|$ is the entity number in SCADA which has the sensitive resources or has the right to read and write the resources; the other is the time cost on building the FSMs of all these entities. One FSM building time overhead is $O(\lg|VMD|)$, which is the course of searching and matching process in the SCADA vulnerability pattern database using the binary search algorithm. Consequently, the time overhead of all the resources FSM building is $O(|R|*\lg|VMD|)$. Thus, the entire time cost of the vulnerability detection is $\max(O(|Entity|), O(|R|*\lg|VMD|))$.

4 SDNVD-SCADA Implementation

In order to implement the SDNVD-SCADA system, the most important thing is how to solve the formalization problems to let the detection process automatically. Therefore, a SCADA security mechanism description language is designed to describe the security mechanism of different entities in SCADA system. A SCADA vulnerability pattern description syntax is described to construct the vulnerability pattern database. And the corresponding SCADA vulnerability detector is presented to make the detection automatically.

4.1 SCADA Security Mechanism Description Language

SDNVD-SCADA provides a formalized approach to describe the security mechanism of different entities at various levels. Thus, SDNVD-SCADA provides a standardized and formalized language to analyze SCADA vulnerabilities.

The language provides the capability to describe the class specification in BNF paradigm syntax with XML format recursively. So, various types of security mechanisms at all levels can find the appropriate location in the SDNVD-SCADA framework. By language decomposes its complexity and highlighting the most essential features, which can effectively improve the analysis capabilities of SDNVD-SCADA model and establish the automated analysis foundation for SCADA vulnerability detection.

The entity security mechanism is divided into three levels: system-level, Internet-level and operation level. Based on the study of various SCADA vulnerability cases and data, we define the vulnerability patterns as the missing of resources confidentiality, integrity and availability.

Definition 1. X denotes an entity set, and R is a resource. If the entity in X cannot access the information on R , then define R for X is **confidential**.

Definition 2. Set X the aggregation of entities, and R is a resource. If all members in X trust R , then R has **integrity** for X .

Definition 3. Set X the aggregation of entities, and R is a resource. If all members of X can access R , then R for X with the **availability**.

System security mechanism involves confidentiality, integrity and availability. Confidentiality means that the security mechanism should prevent the resources from leaking to unauthorized entity. Integrity indicates that the security mechanisms must specify the authorized entities that can modify the resources. The security mechanism to describe the conditions and modes of the resource changes is called integrity strategy. Availability refers to that the security mechanism should describe the resources which must be provided. It defines the resources parameters and the access range.

SCADA security mechanism description language defines some basic elements in SCADA interaction, as shown in Table 1. The main contents are divided into two categories: the one is the basic definitions related to the interactive entities and processes; the other is the basic definitions of security mechanisms, which include resource access privileges, resource security features, and its security patterns.

Table 1. The elements definition of SCADA security mechanism description language

Elements Definition	
<i>Entity ID</i>	<i>Entity identification</i>
<i>Duration</i>	<i>Duration for an interaction process</i>
<i>ResName</i>	<i>Resource identification</i>
<i>Data</i>	<i>Data type of resource</i>
<i>Information</i>	<i>Information type of resource</i>
<i>Service</i>	<i>Service type of resource</i>
<i>Confidentiality Sensitive</i>	<i>Be sensitive for confidentiality</i>
<i>Confidentiality Insensitive</i>	<i>Be insensitive for confidentiality</i>
<i>Integrity Sensitive</i>	<i>Be sensitive for integrity</i>
<i>Integrity Insensitive</i>	<i>Be insensitive for integrity</i>
<i>Availability Sensitive</i>	<i>Be sensitive for availability</i>
<i>Availability Insensitive</i>	<i>Be insensitive for availability</i>
<i>t_Readable</i>	<i>Resource readable</i>
<i>t_Writable</i>	<i>Resource writable</i>
<i>t_Executable</i>	<i>Resource executable</i>
<i>Authorization Pattern</i>	<i>Entity authorization pattern, including Central pattern and distributed pattern, etc.</i>
<i>Encryption Pattern</i>	<i>Entity encryption pattern, including symmetrical pattern and dissymmetrical pattern, etc.</i>
<i>Authentication Pattern</i>	<i>Entity authentication pattern, including certification pattern and password pattern, etc.</i>

With the syntax definition above, SDNVD-SCADA provides a specification to describe the security mechanisms at different levels by dint of the BNF description methods. In BNF, “::=” denotes “definition”, “|” means “or”, angle brackets “<>” refers to a non-terminal symbol. The so-called non-terminal symbol is some abstract concept in language, and the terminal symbol is that can directly appear in the language.

Table 2 lists the descriptions for some security mechanisms. System level mechanism is mainly concerned about the security mechanisms used by the operating system. Interaction level mechanism is focused on the network layer and application inter-connection layer. Operation level security mechanisms is mainly used to describe the security mechanisms in business processes, including the information of released resources and its authorization, encryption and other information. If there is a new security mechanism, it can be added according to the syntax specifications.

4.2 SCADA Vulnerability Pattern Description

While the manifestations of vulnerabilities are very wide, their fundamental connotation just refers to that the confidentiality, integrity or availability of a resource is compromised. Thus, from the perspective of resource state, a vulnerability can be denoted as follows:

$$\langle \text{Vulnerability} \rangle ::= R_{\neg \text{Confidentiality} | \neg \text{Integrity} | \neg \text{Availability}}$$

Table 2. The syntax of SCADA security mechanism description language

Syntax of SCADA Security Mechanism Description
$\langle \text{Security Mechanism} \rangle ::= \langle \text{System Level Mechanism} \rangle \mid \langle \text{Interaction Level Mechanism} \rangle \mid \langle \text{Operation Level Mechanism} \rangle$
$\langle \text{System Level Mechanism} \rangle ::= \text{Entity ID} \mid \langle \text{Neighbor} \rangle \mid \text{Authentication Pattern} \mid \text{Authorization Pattern} \mid \text{Encryption Pattern} \mid \dots$
$\langle \text{Neighbor} \rangle ::= \langle \text{Entity} \rangle^*$
$\langle \text{Entity} \rangle ::= \text{PLC} \mid \text{MTU} \mid \text{RTU} \mid \text{Master} \mid \text{Switch Node} \mid \text{External Node} \dots$
$\langle \text{Interaction Level Mechanism} \rangle ::= \langle \text{Interaction List} \rangle \mid \dots$
$\langle \text{Interaction List} \rangle ::= \langle \text{Interaction} \rangle^*$
$\langle \text{Interaction} \rangle ::= \langle \text{Src} \rangle \langle \text{Des} \rangle \langle \text{Path} \rangle \langle \text{Msg} \rangle \langle \text{Event} \rangle \text{Duration}$
$\langle \text{Src} \rangle ::= \text{PLC} \mid \text{MTU} \mid \text{RTU} \mid \text{Master} \mid \dots$
$\langle \text{Des} \rangle ::= \text{PLC} \mid \text{MTU} \mid \text{RTU} \mid \text{Master} \mid \dots$
$\langle \text{Path} \rangle ::= \langle \text{Src} \rangle \text{Swith Node}^* \langle \text{Des} \rangle$
$\langle \text{Msg} \rangle ::= \text{ResName}^+$
$\langle \text{Event} \rangle ::= \text{Read} \mid \text{Write} \mid \text{Copy} \mid \text{Cut} \mid \text{Create} \mid \text{Delete} \mid \dots$
$\langle \text{Operation Level Mechanism} \rangle ::= \langle \text{Resource List} \rangle \mid \langle \text{Authorization List} \rangle \mid \dots$
$\langle \text{Resource List} \rangle ::= \langle \text{Resource} \rangle^*$
$\langle \text{Resource} \rangle ::= \text{ResName} \langle \text{ResType} \rangle \langle \text{ResFeather} \rangle$
$\langle \text{ResType} \rangle ::= \text{Data} \mid \text{Information} \mid \text{Service} \mid \dots$
$\langle \text{ResFeather} \rangle ::= \text{Confidentiality Sensitive} \mid \text{Confidentiality Insensitive} \mid \text{Integrity Sensitive} \mid \text{Integrity Insensitive} \mid \text{Availability Sensitive} \mid \text{Availability Insensitive}$
$\langle \text{Authorization List} \rangle ::= \langle \text{AuObject} \rangle \langle \text{AuResource} \rangle \langle \text{AuType} \rangle$
$\langle \text{AuObject} \rangle ::= \text{Entity}^*$
$\langle \text{AuResource} \rangle ::= \text{ResName}$
$\langle \text{AuType} \rangle ::= R \mid W \mid E \mid RW \mid RE \mid WE \mid RWE$
$\langle R \rangle ::= t_Readable$
$\langle W \rangle ::= t_Writable$
$\langle E \rangle ::= t_Executable$

The SCADA vulnerability pattern in the database can be expressed as:

$$E \wedge \neg B \rightarrow V \rightarrow A$$

It indicates that when the event E occurs, if the security mechanism B is invalid or missing, the vulnerability V will appear which may trigger attacks A . The event E is an interactive process, and is represented with a quad-ruple $f(\text{Src}, \text{Des}, \text{ResName}, \text{Action})$ to abstract the key properties of a resource. Action includes the actions of read, write, copy, cut, create, delete, and so on. When a resources is identification or encryption key, Action can include the identity and key creation, distribution and destroy. For example, if a user X requests resource R from server Y , event E can be denoted as $E = [X, Y, R, \text{Read}]$.

Security mechanism B is applied to the resource ResName . As the entity interaction happens between Src and Des , B can be expressed by a triple consisting of resources, methods, and entities, i.e. $B = g(\text{Entity}, \text{ResName}, \text{Mechanism})$, $\text{Entity} \in [\text{Src}, \text{Des}]$, wherein mechanism may cover authorization, encryption, authentication, etc. If the

resource $ResName$ is identity, key or other special types, $Mechanism$ can include the key length, identity and encryption mode. As a demonstration, the server Y provides authorization mechanism B to its resource R , which can be expressed as $B = [Y, R, authorization]$.

The vulnerability V is denoted as seven kinds of forms enumerated as follows:

$$V = [(\neg Confidentiality, Integrity, Availability), (Confidentiality, \neg Integrity, Availability), (Confidentiality, Integrity, \neg Availability), (\neg Confidentiality, \neg Integrity, Availability), (\neg Confidentiality, Integrity, \neg Availability), (Confidentiality, \neg Integrity, \neg Availability), (\neg Confidentiality, \neg Integrity, \neg Availability)].$$

Attack A which may be incurred by vulnerability V includes all SCADA possible attacks, such as Sybil attack, denial of service attack, middle-person attack, etc.

For example, with the definitions above, when a client accesses the resources published by a server, if the server does not provide the authorization mechanism for resource accessing, the resources confidentiality may be compromised. This scenario can be represented in the following manner:

$$V1 : [USER, RP, R, Read] \wedge \neg[RP, R, authorization] \rightarrow \\ \neg Confidentiality \rightarrow Information\ leakage$$

When the server's resources are to be back upped to other servers, if there is no identity authentication for these servers, the resources integrity may not be guaranteed and Sybil attack perhaps takes place. This can be described with the following expression:

$$V2 : [RP1, RP2, R, Copy] \wedge \neg[RP1, R, ID\ Authentication] \rightarrow \\ \neg Integrity \rightarrow Sybil\ Attack$$

4.3 SCADA Vulnerability Detector

In order to characterize the resource states changed with the SCADA entity interaction process, SDNVD-SCADA introduces a finite-state machine $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, wherein:

- $Q = \{q_0, q_1, \dots, q_8\}$ is the collection of resource security state;
- $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_n\}$ is the collection of input events;
- $\Delta = \{a_0, a_1, \dots, a_n\}$ is the collection of output events;
- $\delta: Q \times \Sigma \rightarrow 2^Q$ is the state transformer function;
- $\lambda: Q \times \Sigma \rightarrow \Delta$ is the output function;
- $q_0 \in Q$ is the resource initiate security state.

$Q = V \vee (Confidentiality, Integrity, Availability).$

Σ and Δ are the collection of events E .

$\delta: Q \times \Sigma \rightarrow 2^Q$ is the state transformer function,

$\lambda: Q \times \Sigma \rightarrow \Delta$ is the output function.

These two functions provide the basis rules for resources security state transformation. The definitions of δ and λ refer to the SCADA vulnerability pattern database. If the input events and the resource security mechanisms will lead to a vulnerability, LDS-IVDM will work on the resource security attributes of furthermore. If the resource is sensitive to this vulnerability, the security state of resource will switch to a fragile state; otherwise, the resource security state will remain, namely:

$$\delta, \lambda(\sigma_i, q_i) = \begin{cases} v, v \in V, & \text{if } \sigma_i \in E \wedge \text{SecurityMechanism} \in \neg B \wedge v \notin \text{ResFeather} \\ \text{else} & (\text{Confidentiality, Integrity, Availability}) \end{cases}$$

5 Example

In this section, we will show how to use SDNVD-SCADA to find vulnerabilities in SCADA by a typical example. Figure 3 exhibits the network topology of a SDN-enabled SCADA system used in the power grid. A SDN controller is deployed to control all the information transmission in the network. The entities in the SCADA include the OPC Server, PLC, MTU, RTU, Interface terminal etc. They transmit the control and data information between each other. For example, RTU uploads its collected information R_i about the power nodes to the OPC Server, PLC node read the industrial data R_a from the OPC Server, and PLC node transfer the control command R_c to the OPC Server.

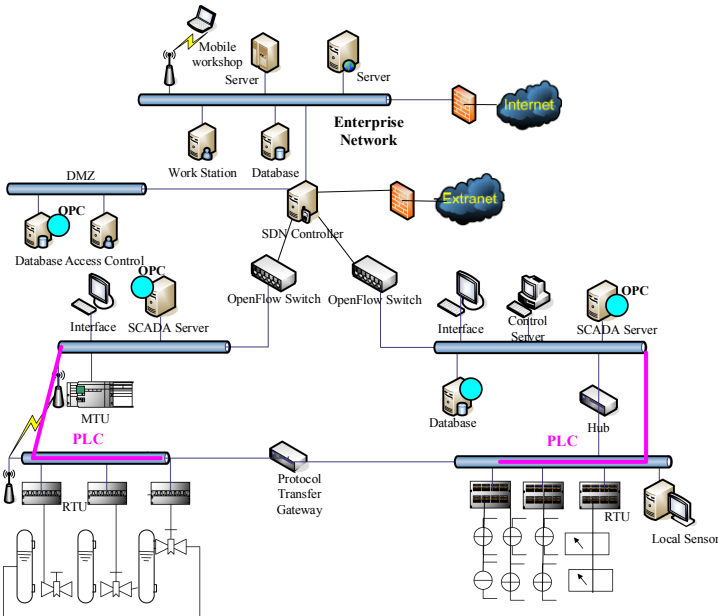


Fig. 3. Network topology of a typical SDN-enabled SCADA system used in the example. A SDN controller is deployed to control all the information transmission in the network. The entities in the SCADA include the OPC Server, PLC, MTU, RTU, Interface terminal etc.

The resource R_i , R_a and R_c are all sensitive the confidentiality, integrity and availability. Any action that may destroy the C-I-A of them is a vulnerability to the SCADA system. According to the applicable reality, R_i is a part of R_a , because OPC Server collect the R_i from all RTUs to construct the R_a . That means R_i has the same security feature of R_a (Fig. 4).

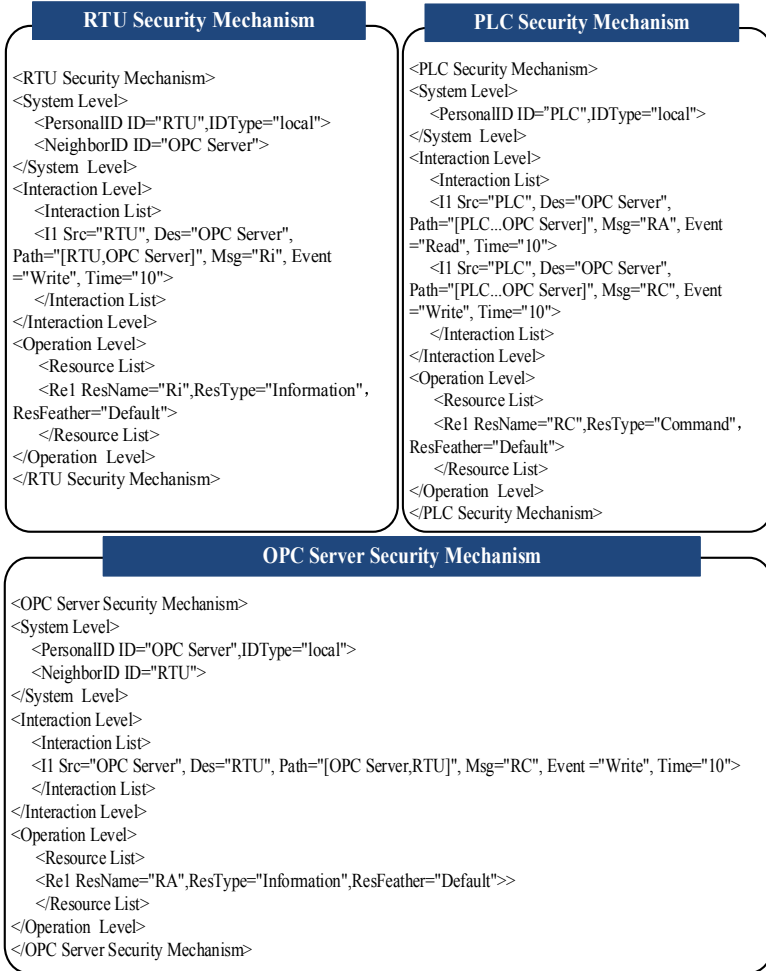


Fig. 4. Security mechanism description of SCADA entities. The security mechanisms of RTU, PLC and OPC are described in SCADA security mechanism description language.

The process proceeding of SDNVD-SCADA is illustrated as follows. Firstly, SDNVD-SCADA pre-installed on the SDN controller analyzes the security mechanism files submitted by RTU, PLC and OPC Server. Secondly, it extracts the related mechanism of R_a and R_c . At the same time, The SDN controller monitors all the transformation in the network and builds the FSMs of R_a and R_c security states, which are illustrated

in Fig. 5. The events related to R_a and R_c include *OPC Server Create Ra*, *OPC Server Write Ra from RTU*, *PLC Read Ra from OPC Server*, *PLC create Rc*, *PLC Write Rc to OPC Server*, *OPC Server Write Rc to RTU*. Finally, SDNVD-SCADA detects the possible vulnerabilities and work out the potential attacks according to the part in the corresponding items in SCADA vulnerability pattern database. When the states of R_a and R_c transfer to the R2 and R3, they might occur a security problem.

- 1) To the R2 vulnerability state, it may be occurred by the remote anonymous log in, which is be permitted by the default deployment of many SCADA device corporations. So the attackers from the Internet or external network could access the OPC Server CLSID on the OPC Server, and analyze the Controller type and software information of the SCADA system. The confidentiality of the system is broken.
- 2) To the R3 vulnerability state, it may be occurred by the absence of the encryption mechanism in the OPC communication course. After the attackers connect in the SCADA network, it can listen and analyze the important information, or disguise itself as a OPC client and send the fake packets to the OPC server to destroy the accurate control process of SCADA.

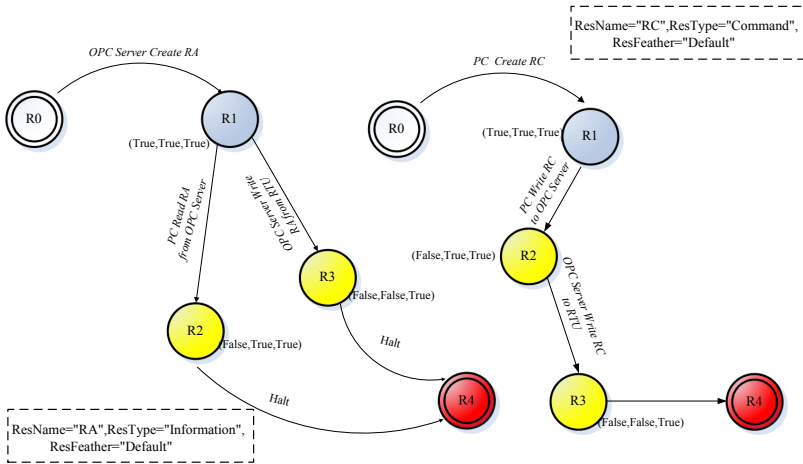


Fig. 5. FSMs of R_a and R_c Resource Security State. When the states of R_a and R_c transfer to the R2 and R3, they might occur a security problem.

6 Conclusion

In this paper, we focus on the security problem in SCADA system and propose a new solution on automatic vulnerability detection by adopting the SDN technology. The SDN controller implemented SDNVD-SCADA can centralized analyze the security mechanisms of important entities in SCADA and monitor all the information transferring about the security-sensitive resources in SCADA. By the formalized security mechanism

description language and SCADA vulnerability pattern database, a finite-state machine can be built of each security-sensitive resource, by which the vulnerable state of resource could be detected by SDNVD-SCADA.

The prototype of SDNVD-SCADA is under construction on one of the popular open source SDN controller, Floodlight v1.0. More SCADA vulnerability pattern should be collected and more experiments should be tested later.

References

1. Gresser, C.H.: Hacking SCADA/SAS systems. In: Seminar at Petroleum Safety Authority Norway (2006)
2. Dong, X., Lin, H., Tan, R., Iyer, R.K., Kalbarczyk, Z.: Software-defined networking for smart grid resilience: opportunities and challenges. In: Cyber Physical System Security (CPSS 2015), Singapore (2015)
3. Irfan, N., Mahmud, A.: A novel secure SDN/LTE-based architecture for smart grid. In: Proceedings of the 2015 IEEE International Conference on Computer and Information Technology, pp. 762–769 (2015)
4. Machii, W., et al.: Dynamic zoning based on situational activities for ICS security. In: Proceedings of the 2015 10th Asian Control Conference (ASCC), pp. 1–5 (2015)
5. Chavez, A., Hamlet, J., Lee, E., Martin, M., Stout, W.: Sandia National Laboratories, network randomization and dynamic defense for critical infrastructure systems, Albuquerque, New Mexico and Livermore, California, USA (2015)
6. Silva, E., Knob, L., Wickboldt, J., Gaspary, L., Granville, L., Schaeffer-Filho, A.: Capitalizing on SDN-based SCADA systems: an anti-eavesdropping case-study. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), Ottawa (2015)
7. Genge, B., Haller, P., Beres, A., Sándor, H., Kiss, I.: Using software-defined networking. *Securing Cyber-Phys. Syst.* 305–329 (2016)
8. Simões, P., Cruz, T., Proença, J., Monteiro, E.: On the use of honeypots for detecting cyber-attacks on industrial control networks. In: Proceedings of the 12th European Conference on Information Warfare and Security, pp. 263–270 (2013)
9. Song, Y., Shin, S., Choi, Y.: Network iron curtain: hide enterprise networks with openflow. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2013. LNCS, vol. 8267, pp. 218–230. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05149-9_14
10. Adrichem, N., Asten, B.J., Kuipers, F.A.: Fast recovery in software-defined networks. In: EWSDN 2014 Proceedings of the 2014 Third European Workshop on Software, Washington (2014)
11. Dorsch, N., Kurtz, F., Georg, H., Hagerling, C., Wietfeld, C.: Software-defined networking for smart grid communications: applications, challenges and advantages. In: IEEE International Conference on Smart Grid Communications (2014)
12. Queiroz, R.M.C.: Integration of SDN technologies in SCADA Industrial Control Networks. Masters' Degree in Informatics Engineering Dissertation, 15 January 2017
13. Rehmani, M.H., Davy, A.: Software defined networks based smart grid communication: a comprehensive survey. [arXiv:1801.04613v4](https://arxiv.org/abs/1801.04613v4) [cs.NI], 27 March 2019
14. Beibei, L.: A Software-Defined Networking (SDN) Assisted Middleware Interconnecting Supervisory Control and Data Acquisition (SCADA) Systems. Thesis of the Degree Master of Science, Arizona State University, August 2018



Optimal Implementation of In-Band Network Management for High-Radix Switches

Jijun Cao^(✉), Mingche Lai, Xingyun Qi, Yi Dai, and Zhengbin Pang

College of Computer, National University of Defense Technology, Changsha, China
{caojijun,laimingche,qixingyun,daiyi,pangzhengbin}@nudt.edu.cn

Abstract. To manage (such as configuring and monitoring) the numerous network chips and its ports efficiently, the in-band management technology is used in the interconnect network of high performance computing systems. However, with the rapid development of network switching chips towards the higher radix, the traditional in-band management implementation of ring structure faces the problem of delay performance scalability. The work proposed two optimized implementation structures for the in-band management, four-quadrant double-layer ring and four-quadrant star ring to solve the problem. The results of resource consumption assessment and delay performance simulation showed that in the high-radix switching chips with 64, 80, 96, 112, 128, 144, and 160 ports, the occupancies of LUT (Look Up Table) resources of the four-quadrant double-layer ring and star ring structures increased by an average of 5.46% and 1.71% compared to the traditional ring structure, respectively. Meanwhile, the occupancies of LUTRAM (Look Up Table memory) resources increased by an average of 30.89% and 21.81%; that of FF (Flip Flop) resources by an average of 3.86% and 0.19%; the forward delay of management packets decreased by 25.75% and 21.81%, respectively. Considering both resource consumption and delay performance, the star ring was an ideal structure to deal with the problem of delay performance scalability among the in-band management structures, which can be applied to realize the in-band management for the higher-radix switching chips in the future.

Keywords: High-radix switching chips · In-band network management · Single ring · Four-quadrant double-layer ring · Four-quadrant star ring

1 Introduction

According to the implementation methods of transmission path of management information (i.e., management path), the interconnect network management of high-performance computing systems is classified into the in-band and out-band management. The path of out-band management is independent of the data path

This work was supported by The National Key Research and Development Program of China (grant 2018YFB0204300).

of the network; however, the management information transmission and network data transmission of the in-band management multiplex the same physical path. Compared with the out-band management, the in-band management has the advantages of high performance, low implementation cost, and maintaining fault consistency. Therefore, it is an essential means to achieve efficient management of the interconnect networks of a high-performance computing system [1].

In the interconnect network of the high-performance computing system, the high-radix switching chips (i.e., switches) reduce the diameter of the network system and the average number of hops for communication between nodes. Therefore, the communication delay between nodes reduces. The switching chip ports increase to reduce the switching nodes in the interconnect networks, thereby improving the reliability of the network system. Also, the high-radix switching chips provide abundant interconnection ports, which provides flexibility for designing the network topology. Networks with redundant links and supporting multi-path routing can improve the availability of the system. Designing a high-performance interconnect network based on the high-radix switching chips has become the mainstream trend. As Kim and Dally predicted more than a decade ago, the mainstream switching chips are developing towards a higher radix [2, 3]. Table 1 shows the high-radix switching chips that have been commercialized or applied to actual systems in recent years.

Table 1. Several typical switching chips of high-radix network.

Switching chips	Year	Radix	Comp.	Ref.
Omni-Path Architecture 4.8 Tbps Switch ASIC	2015	48	Intel	[7, 10]
Centec CTC8096 (GoldenGate)	2016	96	Centec network	[5]
Broadcom StrataDNX TM BCM88690	2017	112	Broadcom	[4]
Broadcom Tomahawk3 BCM56980	2017	128	Broadcom	[4]
Broadcom StrataDNX TM BCM88790	2017	192	Broadcom	[4]
Quantum TM HDR(4x) InfiniBand Switch IC	2019	40	Mellanox	[6]
Spectrum TM -2 Ethernet Switch ASIC	2019	128	Mellanox	[6]

In a word, the in-band management is an important technology for the high-speed interconnect network. However, with the switching chips developing towards higher radix, the present in-band management will face the problem that delay performance is challenging to extend. For the higher-radix switching chips, the work studied the optimal implementation of the in-band management.

The rest of this paper is organized as follows. Section 2 discusses the related works of in-band network management. Section 3 analyzes the problem of delay performance scalability faced by implementing the in-band management of ring structures in higher-radix switching chips. Two optimized structures for the in-band management are presented in Sect. 4. Section 5 assesses the logic resources occupied by the structure implementing new in-band management; meanwhile, Sect. 6 evaluates the delay performance of the new structure implementing the in-band implementation. Finally, Sect. 7 draws relevant conclusions.

2 Related Works

2.1 IB In-Band Management

Mellanox company implemented the InfiniBand high-speed interconnect protocol [8]. Subnet management, the basis for the InfiniBand network interconnection, adopts an autonomous way to discover the subnet topology and deal with its changes. It can ensure the availability of the network without external interferences. The main functions of InfiniBand subnet management include topology discovery, route calculation, and distribution of forwarding tables. The subnet management is mainly completed by the subnet manager (SM) and subnet management agent (SMA). The SM is responsible for controlling and checking the subnet, while the SMA sets and queries the management parameters of each endpoint. SM and SMA send the data packets of subnet management to communicate with each other through the subnet management interface. In the subnet, each managed node has an SMA, but there is usually only one main SM responsible for discovering, configuring, activating, and maintaining the subnet.

2.2 OPA In-Band Management

Intel Corporation proposed and implemented the Omni-Path Architecture (OPA) for interconnect network [9, 10]. OPA adopts the centralized network management, that the network management function is mainly completed by the centralized fabric manager and the management agents distributed in each network component (including switches and host fabric interfaces (HFIs)). Fabric Manager is a soft component of OPA network. Each OPA network can instantiate multiple fabric managers, but during the network initialization process, only one fabric manager is selected as the primary fabric manager. The features of main fabric manager include discovering network topology, providing identification information of network component, calculating and delivering switch forwarding tables, maintaining network management databases, and monitoring network performance and fault status. Network management messages are transmitted by separate virtual channels and buffers, and management messages can be transmitted in flow and non-flow controls. If using the latter, the management messages are allowed to be discarded when the port buffer resources are unavailable during the transmission. The management message of the OPA network adopts directional routing, with the maximum supported number of hops of 64.

2.3 TianHe In-Band Management

Nowadays, the National University of Defense Technology (NUDT) has successfully developed two generations of TianHe supercomputer system, in which the TianHe-2 interconnect network also adopts the in-band management [1]. The essential features of the TianHe in-band management include parameter configuration, status monitoring, failure active reporting, link test, topology discovery, and path tracking. Management server sends management request messages to

the network and receives management response messages or failure active report messages from the network to achieve the in-band management.

TianHe in-band management adopts source routing to transmit packets. The management server contains two routing tables, a forward routing table, and a backward routing table. Each item of the forward routing table indicates the routing field from the management server to certain network chip, while the item of the backward routing table indicates the routing field from certain network chip to the management server. The routing field is expressed as $\langle \text{HopNum}, \text{Hop0}, \text{Hop1}, \text{Hop2}, \dots, \text{Hop}(N-1) \rangle$. Wherein, the HopNum represents the number of valid hops, which is decremented by 1 for each hop. Under normal circumstances, when the HopNum reduces to 0, the packet will reach the destination network chip. During the packet transmitting process, the output port of each hop is designated by Hop0 , Hop1 , \dots , and $\text{Hop}(N-1)$, respectively.

When a management server sends a request packet to a chip, indexed by the serial number of the chip, the management program queries the forward and the backward routing table to obtain the forward and backward routing fields, respectively. Then the program fills the information into the request packet, which is routed to the destination chip according to the forward routing field. Meanwhile, the response packet constructed by the destination chip is routed to the server based on the backward routing field. Failure active report packet is sent to the management server when the chip fails, the routing field of which is determined by the configuration of the chip.

TianHe network implements nine virtual channels (VC) based on credit flow-control [11], namely VC_0 – VC_8 . VC_8 is used separately for the management packets, which are transmitted on the network link by a credit-based flow control method. Management packets, including management request packets, management response packets, and fault active report packets, are of fixed size. It is comprised of 6 flits, which is the smallest unit of the flow control. The size of each flit is 256 bits.

3 Scalability Problem of In-Band Management Ring

3.1 Ring Structure of In-Band Management

TianHe network mainly includes two types of network chips, interface chip and switching chip. For in-band management, the former is used to convert the management descriptors in the descriptor queue (DQ) to management request packets and process management request packets. Meanwhile, it has the following functions such as managing the generation of management response packets and failure active report packets, receiving the management response packets and failure active report packets, and writing the above packets into mini-packet queue (MPQ). The upper management program sends a management request packet by writing the management descriptor to the DQ, and then receives a response by reading the management response or failure active report packet from the MPQ. The in-band management of the interface chip is mainly realized by the in-band management agent (INM Agent).

Network switching chip is to process the management request packets, manage the generation of management response and failure active report packets, and control the packet routing. Figure 1 shows the data path structure of the in-band management for register access in the switching chips. In-band management agent (INM Agent) and CSR (Control and Status Registers) controllers are the main functional modules of a network management engine (NM Engine). INM Agent is connected to each logic port of the chip (specifically including PCS and Link-Layer) to form the ring channel of the in-band management. Meanwhile, the CSR Controller is connected to three sets of the CSR-ring channels, respectively, including underlying SerDes path, port path, and Tile path.

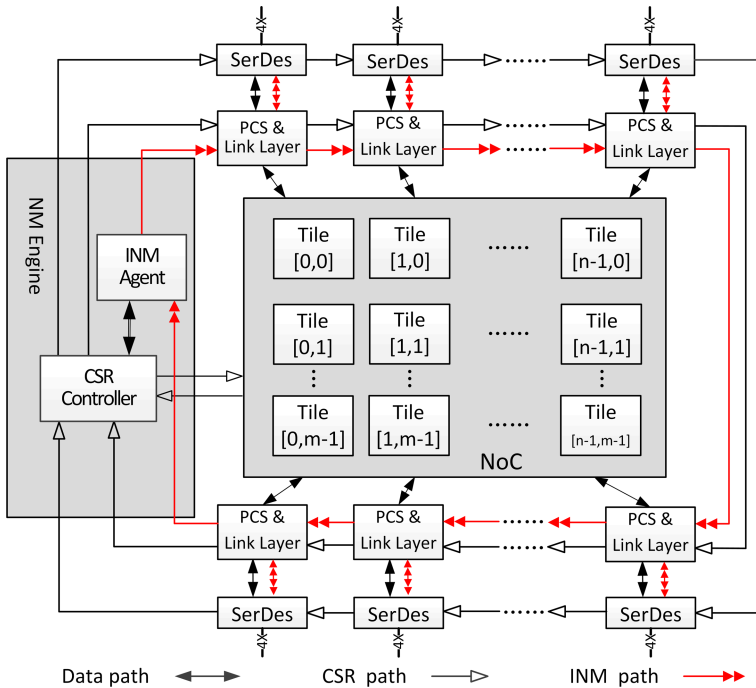


Fig. 1. Data path structure of in-band management (register access) in switching chips.

CSR Controller receives the register operating commands from the INM Agent, and then injects the commands into a CSR ring. The register operating command is transmitted sequentially on the ring, each module of which recognizes the operating command according to the registered addresses. Therefore, the information is read and written to the corresponding register. In case of a register read command, the CSR Controller needs to wait for the data from the CSR ring and return it to the INM Agent in response. For the register write command, it does not need to return a write completion response. Taking the

register read access as an example, when the INM Agent receives a management request packet with the chip as access target from the ring channel of the in-band management, the INM Agent will convert the request into a register read command for the CSR Controller to start the on-address read aiming at the corresponding register. Then the INM Agent waits for the CSR Controller to return the register value, thus generating a management response packet and finally injecting the management response into the data path of the in-band management.

The length of each flit of management packet is 32 bytes. The width of the ring channel is designed to be 40 bits to save wire resources and reduce the difficulty of the backend design of chips. The single-clock data is called the in-band-management ring flit. Therefore, the link flit is divided into seven ring flit, while each management packet into 42 ring flits. Moreover, to facilitate the routing of ring flits, a ring flit is added as a head ring flit especially, the format of which is defined as $\langle DestRSID, RSHopCnt, PkgType \rangle$. $DestRSID$ (7bit) indicates the serial number of the switch chip port that the ring flit leaving from. $RSHopCnt$ (6bits) shows the number of hops in the ring reduced by 1 for each port transmitted in the management ring, with an initial value of 63. Meanwhile, $PkgType$ (2bits) represents the packet types, that is, management request packet, management response packet and failure active report packet. They are used to implement priority scheduling policies.

3.2 Problem of Delay Performance Scalability

The the relay station (RS) module is implemented and instantiated in each network port logic to realize the routing of the in-band management packets in switching chips. Figure 2 shows the in-band management ring constructed by these connected modules. RS module was implemented to realize the 2×2 switching. The inputs of the two directions were the data from the upper-level RS of the in-band management ring and the management packet data received from the physical layer, respectively. Meanwhile, the outputs were the data transmitted to the next level RS of the in-band management ring and the management packet data sent by the current RS to the physical layer of the port. RS adopted the input queue (IQ) switching, which was implemented by the register array (RA) with 64-bit depth and 40-bit width. The management packets received from the physical layer needed to be width-converted and then stored into the FIFO (RECV FIFO). The sub-modules of the RS data receiver and transmitter for the data-width transformation were RS_RX and RS_TX, respectively.

It was noteworthy that INM Agent instantiated the RS⁺ module. The switching structure of the RS⁺ module was basically the same as that of RS. However, there were two differences between them. (1) IQ on the Chain was deep (using 1024 deep RA in CHAIN FIFO design), so more management packets were cached, which limit the number of concurrent management packets in the network for deadlock avoidance. (2) The two data interfaces of RS_RX and RS_TX were connected to the INM Agent, respectively, instead of the physical logic of the ports. Supposing that the number of network ports of switching chip was N ,

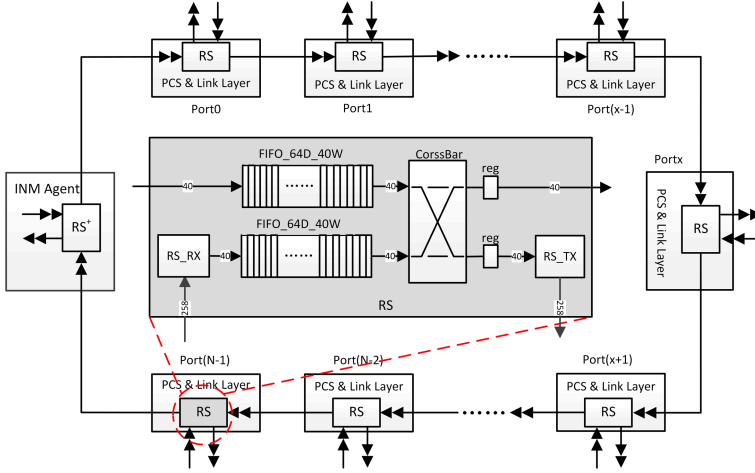


Fig. 2. Implementation of data path of the in-band management in the switching chip.

expressed as P_0, P_1, \dots, P_{N-2} , and P_{N-1} , then there were N RS and one RS^+ in the switching chip. The management packet routing and the number of hops in the loop are as following cases (See Table 2, of which A means INM agent).

According to the Table 2, the routing of the in-band management packets in the switching chip was mainly divided into three situations. (1) The management request for accessing the chip was routed to INM Agent through the in-band management ring. (2) The in-band management response or failure active report packets generated by this chip was routed from the INM Agent to the output port through the in-band management ring. (3) The management packets transferred through the switching chip entered the network port and then was routed to the output port through the in-band management ring. Except for the third case that the input and output port were the same, and the number of hops of the ring flit was 1, the maximum number and average number of hops are linearly related to the number of the switch chip ports.

Table 2. Switching hops analysis of in-band management ring.

Routing Cases	Routing Path	Min Hops	Max Hops	Avg. Hops
$P_i \rightarrow A$	$\rightarrow P_i \rightarrow P_{i+1} \rightarrow \dots \rightarrow P_{N-2} \rightarrow P_{N-1} \rightarrow A \rightarrow$	1	N	$(N+1)/2$
$A \rightarrow P_j$	$\rightarrow A \rightarrow P_0 \rightarrow P_1 \dots \rightarrow P_{j-1} \rightarrow P_j \rightarrow$	1	N	$(N+1)/2$
$P_i \rightarrow P_j, i < j$	$\rightarrow P_i \rightarrow P_{i+1} \dots \rightarrow P_{j-1} \rightarrow P_j \rightarrow$	1	N	$(N+1)/2$
$P_i \rightarrow P_j, i = j$	$\rightarrow P_i \rightarrow$	1	1	1
$P_i \rightarrow P_j, i > j$	$\rightarrow P_i \rightarrow P_{i+1} \dots \rightarrow P_{N-1} \rightarrow A \rightarrow P_0 \dots \rightarrow P_{j-1} \rightarrow P_j \rightarrow$	2	$N+1$	$(N+3)/2$

As the switching chip continues to develop toward higher radix, the number of hops in the management ring flit routing increases. Meanwhile, the average number of hops increases rapidly with the number of ports. It results in a continuously increasing routing time of management packets in the switching chip,

and ultimately increases the time from sending management request to receiving response. Therefore, the efficiency of the in-band management is affected, which is called the *Problem of Delay Performance Scalability of the In-Band Management Ring*.

4 The Proposed Structures for In-Band Management

The work referred to the traditional ring structure of the in-band management path as Single Ring structure. For convenience, taking 64-port switch chip as an example, Fig. 3(a) shows the large single ring structure of the chip. Wherein, the circle numbered $i(0 \leq i \leq 19)$ indicated the RS module embedded in the port i of switching chip, and the circle numbered M indicated the RS⁺ module embedded in the INM agent of switching chip.

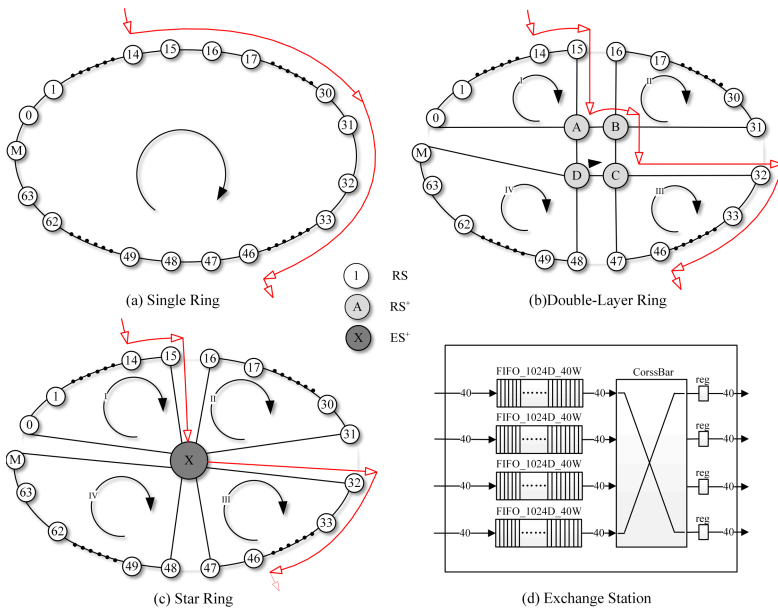


Fig. 3. Three topology structures for in-band management ring.

For the problem of delay performance scalability of the in-band management ring in the high-radix switching chips, the data-path structure of the single large ring must be changed. However, if using the common multi-dimensional interconnect topologies such as Mesh and Torus, the wiring between ports became more complex, thus bringing challenges to the overall architecture and back-end design of switching chips. For that reason, the work proposed two structures of in-band management: double-layer ring and star ring.

4.1 Double-Layer Ring Structure

Figure 3(b) shows the double-layer ring structure with 64 ports. Sixty-four RS and one RS^+ are divided into four quadrants (quadrant I, II, III, and IV) by the distribution of relative positions, among which RS^+ is placed in the fourth quadrant. An RS^+ node is added to quadrants I, II, III, and IV, respectively, named A, B, C, and D (called the root RS^+ of the quadrant). RS and RS^+ in each quadrant are connected according to the number to form the first-dimensional ring structure. Meanwhile, the transmission and receiving ports of the four root RS^+ are connected end-to-end to form the second-dimensional ring structure.

For the management packets switching with source and destination in the same quadrant, if the source number was less than or equal to the destination number, the packets should not go through the root RS^+ . For the management packets switching with source and destination in different quadrants, the packets must go through the root RS^+ , at least two, and at most four root RS^+ . In this structure, the data needed to be converted from ring flit to flit when entering the RS^+ ring from the RS ring, while the data was in the store-and-forwarded mode in the RS^+ ring.

4.2 Star Ring Structure

Figure 3(c) shows the star ring structure with 64 ports. The star ring and double-layer ring structures were consistent in quadrant division design. Their difference was that the double-layer ring structure formed by adding four roots RS^+ to construct a ring and connecting each root RS^+ to RS in this quadrant to form a ring structure. However, the star ring structure connected the RS in the four quadrants by adding one 4×4 full switching structure.

Corresponding to the management packets switching of the double-layer ring structure, for the management packets switching with source and destination in the same quadrant, if the source number was less than or equal to the destination number, the packets should not go through the switching structure of X. However, for the management packets switching with source and destination in different quadrants, the packet must go through ES^+ (Exchange Station).

Similar to the core switching in RS, the ES^+ used the IQ (Input Queue) switching; meanwhile, the queue was implemented by 1,024 deep 40-bit wide RA (Register Array). RS realized the 2×2 full switching, while the ES^+ implemented the 4×4 full switching. RS not only realized the switching of ring flits but also achieved the data width conversion between ring flit and flit. However, ES^+ only realized the data switching of the ring flits instead of the data-width conversion between the ring flit and flit. Figure 3(d) shows the structure of ES^+ . Compared with the traditional single ring structure, the double-layer ring and star ring structures divided the one-dimensional ring into four quadrants. One-dimensional ring routing was used in each quadrant; however, the flit routing of management ring was implemented between quadrants through a two-dimensional ring or switching structure, respectively. As a result, the number of hops in the routing of management ring flit reduced overall. For example, the management packet was imputed from port

14 of the switching chip and outputted from port 46. (1) In the single ring structure, the routing process of ring flit was $RS(14) \rightarrow RS(15) \rightarrow \dots \rightarrow RS(30) \rightarrow RS(31) \rightarrow \dots \rightarrow RS(45) \rightarrow RS(46)$, with a total of 33 loop hops required. (2) In the newly proposed double-layer ring structure, the routing process of ring flit was $RS(14) \rightarrow RS(15) \rightarrow RS^+(A) \rightarrow RS^+(B) \rightarrow RS^+(C) \rightarrow RS(32) \rightarrow RS(33) \rightarrow \dots \rightarrow RS(45) \rightarrow RS(46)$, with a total of 21 ring hops required. (3) In the newly proposed star ring structure, the routing process of ring flit was $RS(14) \rightarrow RS(15) \rightarrow ES^+(X) \rightarrow RS(32) \rightarrow RS(33) \rightarrow \dots \rightarrow RS(45) \rightarrow RS(46)$, with a total of 19 ring hops required.

5 Resource Assessment

For the high-radix switching chip with N ports, both the traditional single ring structure and the newly proposed double-layer ring structure and star ring structure contained an INM Agent and $N-1$ RS modules. The difference was that the double-layer ring structure additionally contained four RS^+ modules and the star ring structure additionally contained one ES^+ module. Table 3 shows the overall resource occupancy of sub-modules of the three ring structures.

Table 3. Overall resource occupancy of sub-modules of in-band management ring.

Structure	Resources
Single ring	$\text{Res}(\text{INM Agent}) + (N-1) \cdot \text{Res}(\text{RS})$
Double-layer ring	$\text{Res}(\text{INM Agent}) + (N-1) \cdot \text{Res}(\text{RS}) + 4 \cdot \text{Res}(\text{RS}^+)$
Star ring	$\text{Res}(\text{INM Agent}) + (N-1) \cdot \text{Res}(\text{RS}) + \text{Res}(\text{ES}^+)$

To further evaluate the logic resources occupied by the modules, the RTL code of the in-band management modules were synthesized by the FPGA design platform. The synthesizer tool was the Vivado v2018.2 (64-bit) developed by Xilinx Corporation [12], and the synthesis target device selected the Virtex UltraScale series of xcvu440-flga2892-2-e. Meanwhile, the synthesis adopted the default policy: Vivado Synthesis Defaults. The logical resources in FPGA included CLB (Configurable Logic Block), LUT (Look Up Table), Register, CARRY8 (a fast carry logic for performing addition and subtraction), F7 Mux and F8 Mux (two multifunctional multiplexers), LUTRAM (Look Up Table memory), and FF (Flip Flop).

Table 4 shows the resource occupancy for each sub-module of the in-band management. INM Agent occupied the most resource among these four basic logic modules. It contained the logic for register access, E^2 prom access and Flash access; however, the RS, RS^+ , and ES^+ only contained the logic for data format conversion and packet switching. Under the existing technology and process for ASIC, the chip with no less than 64 ports was usually called a high-radix switching chip. For this reason, the in-band management of high-radix switching

Table 4. Resource occupancy for each sub-module of in-band management.

Structure type	CLB LUTs	CLB registers	CARRY8	F7 Muxes	F8 Muxes
INM Agent	6459	10345	49	185	6
RS	1971	4167	17	170	0
RS ⁺	2840	4217	22	251	0
ES ⁺	3594	883	14	240	91

chips with 64, 80, 96, 112, 128, 144, and 160 ports was used for the resource assessment in the work. The synthesizing environment was the same as that of the sub-modules of the in-band management. Table 5 shows the total resource occupancy of the modules of the in-band management of typical high-radix switching chips, only counting the usage of three main resources, LUT, LUTRAM, and FF.

Table 5. Resource occupancy of the modules of the in-band management of typical high-radix switching chips.

Structure type	LUTs			LUTRAMs			FFs		
	SingleR ^α	DoubleLR ^β	StarR ^γ	SingleR	DoubleLR	StarR	SingleR	DoubleLR	StarR
64	132605 (1.0000)	143905 (1.0852)	136136 (1.0266)	6832 (1.0000)	10096 (1.4778)	9136 (1.3372)	277034 (1.0000)	293896 (1.0609)	277879 (1.0031)
80	164140 (1.0000)	175565 (1.0696)	167708 (1.0217)	8368 (1.0000)	11632 (1.3901)	10672 (1.2753)	343706 (1.0000)	360574 (1.0491)	344561 (1.0025)
96	195676 (1.0000)	207084 (1.0583)	199221 (1.0181)	9904 (1.0000)	13168 (1.3296)	12208 (1.2326)	410378 (1.0000)	427245 (1.0411)	411226 (1.0021)
112	227212 (1.0000)	238652 (1.0503)	230791 (1.0158)	11440 (1.0000)	14704 (1.2853)	13744 (1.2014)	477050 (1.0000)	493918 (1.0354)	477893 (1.0018)
128	258743 (1.0000)	270062 (1.0437)	262284 (1.0137)	12976 (1.0000)	16240 (1.2515)	15280 (1.1776)	543717 (1.0000)	560585 (1.0310)	544564 (1.0016)
144	290285 (1.0000)	301709 (1.0394)	293873 (1.0124)	14512 (1.0000)	17776 (1.2249)	16816 (1.1588)	610394 (1.0000)	627256 (1.0276)	611243 (1.0014)
160	321820 (1.0000)	333221 (1.0354)	325388 (1.0111)	16048 (1.0000)	19312 (1.2034)	18352 (1.1436)	677066 (1.0000)	693934 (1.0249)	677911 (1.0012)

^αSingleR is Single Ring; ^βDoubleLR is Double-Layer Ring; ^γStarR is Star Ring.

The resource assessments based on FPGA design platform showed that: (1) In the high-radix switching chips with 64, 80, 96, 112, 128, 144, and 160 ports, the LUT resource occupancy of the four-quadrant double-layer ring and star ring structures increased by an average of 5.46% and 1.71% compared to simple ring structure, respectively. Meanwhile, the occupancy of the LUTRAM resource increased by an average of 30.89% and 21.81%; that of FF resource by an average of 3.86% and 0.19%, respectively. (2) Compared with simple ring structure, LUTRAM \gg LUT > FF was the rank order of an increasing proportion of resources occupied by four-quadrant double-layer ring and star ring structures. In other words, LUTRAM was the main factor for the increased resource. (3) Compared with the simple ring structure, the increasing ratio of LUT, LUTRAM, and FF resources occupied by the four-quadrant double-layer ring and star ring structure gradually decreased with the increasing number of ports.

6 Performance Evaluation

The performance of the in-band management mainly refers to the delay in accessing CSR, E²prom, or Flash through the in-band management path. In the high-performance computing systems, the in-band accessing delay refers to the completion time of the in-band network that managing one request-response transaction. The work [1] has modeled the in-band accessing delay, the approximate model of which was that: *in-band accessing latency = the processing delay of management packet + (the number of hops + 1) × the single-hop two-way average transmission delay of the management packet*. When considering the difference between the delay of management request and response packet in chips, the more accurate model should be that: *in-band accessing delay = the processing delay of management packet + the switching and transmission delay of the management request packet in each hop + the switching and transmission delay of a management response packet in each hop*. The switching and transmission delay of management packet at each hop included two parts. One was switching delay of the packet from source port to a destination port on the in-band ring. The other was the sending delay and receiving delay of management packet that sequentially passing through physical layer, coding sub-layer, and link layer of network.

The work mainly studied the optimized implementation of the in-band management of the high-radix switching chips, thus evaluating the switching delay from the source port to the destination port of management packets in the three in-band management rings. The next section analyzed the average switching delay of the in-band management packets and then tested the switching delay through simulation.

6.1 Theoretical Analysis

Similar to the average delay analysis in Table 2, the research object of theoretical analysis is the average switching delay of the three ring structures of the in-band management (Port_{*i*} → Port_{*j*}, *i* ≠ *j*). Meanwhile, the work analyzed the average switching delay of the in-band requests for the chip (Port_{*i*} → INM Agent) and the average switching delay of the in-band response from the chips (INM Agent → Port_{*j*}). In the analysis, it was supposed that *N* is the number of network ports of the switching chips; *D* is the single-hop transmission delay of the in-band ring; *F* is the single-hop transmission delay of the RS⁺ ring. Table 6 shows analysis results.

Table 6. Theoretical analysis of average switching delay of three ring structures.

Structure Type	Port _{<i>i</i>} →INM Agent	INM Agent→Port _{<i>j</i>}	P _{<i>i</i>} →P _{<i>j</i>} , <i>i</i> ≠ <i>j</i>
Single Ring	$\approx D \cdot (N/2 + 1)$	$\approx D \cdot (N/2 + 1)$	$\approx D \cdot (N/2 + 1)$
Double-Layer Ring	$\approx D \cdot (N \cdot 5/16) + 3 \cdot F$	$\approx D \cdot (N/8) + 2 \cdot F$	$\approx D \cdot (N \cdot 7/32) + 2 \cdot F$
Star Ring	$\approx D \cdot (N \cdot 5/16 + 1)$	$\approx D \cdot (N/8 + 1)$	$\approx D \cdot (N \cdot 7/32 + 1)$

The theoretical analysis result shows that (1) compared with the single ring structure, the average switching delay of the star ring structure was significantly reduced. (2) Since the data in the four-quadrant double-layer ring adopted the SAF data transmission scheme in the RS^+ ring, which resulted in $F > D$, the average switching delay of double-layer ring structure could be higher than that of single ring structure when N was small. However, with N increasing, the average switching delay of the double-layer ring would be lower than that of a single ring structure. (3) By comparing these three structures, the four-quadrant star ring structure has the best average switching delay performance. (4) The average switching delay of the three structures are all increased linearly with the increased network ports N , but the rates of growth are different.

6.2 Simulation Analysis

To further study the average switching delay of the three ring structures of the in-band management, the RTL implementation of the in-band management ring was regarded as DUT (Device Under Test). Also, the work used the SystemVerilog language to construct a unified performance test environment, and used the VCS of Synopsys corporation [13] for simulation. The DUT was running at 800 MHz according to the current technological level for ASIC. The configurable parameters of the test environment are as follows: (1) the types of the three ring structures of the in-band management; (2) the number of network ports for switching chip, N valued 64, 80, 96, 112, 128, 144, and 160 during the test; (3) packet types: the in-band request packet for the chip (represented by Req.), the in-band response packet from the chip (represented by Ack.), and the in-band packet forwarded by this chip (represented by Forward).

The actual system usually adopts the centralized network management based on in-band, and in most cases, the in-band network management uses the serial request-response. Therefore, the simulation environment in the work used the serial mode when injecting the management packets into DUT, that was, sending the next request packet to the DUT after receiving the previous response packet. Theoretical analysis showed that the switching delay was linearly related to the number of ring hops. To obtain the average switching delay of packets, the work used the RR (Round-Robin) to change the input port and output port of management packets and ensured fair coverage to the combinations of all source ports and destination ports. Finally, it counted the minimum delay, the arithmetic average delay, and a maximum delay of each measurement.

Figure 4 delay of the management packet of the three ring structures. Comparing Table 6 and Fig. 4, the test results and theoretical analysis results were consistent. The following conclusions can be further obtained: (1) The average request delay of double-layer ring structure was higher than that of single ring structure when $N = 64$. However, as N increased, the average request delay of the double-layer ring structure was lower than that of the single ring structure. (2) According to the data fitting results, the parameter value in the theoretical analysis of the average delay was approximately $D \simeq 10$ ns and $F \simeq 60$ ns. (3) In the high-radix switching chips with 64, 80, 96, 112, 128, 144, and 160 ports,

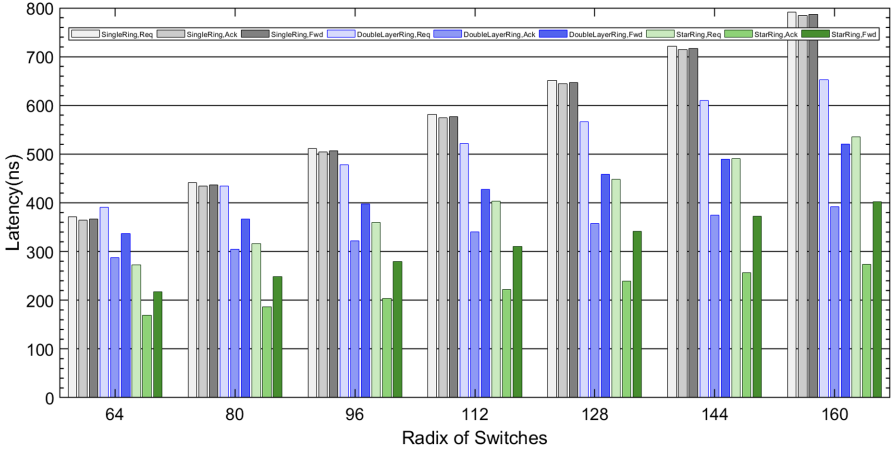


Fig. 4. Simulation results of mgmt packet switching delay of the three ring structures.

compared with single ring, the average switching delay of the four-quadrant double-layer ring and star ring reduced by an average of 25.75% and 62.25%, respectively.

7 Conclusions

In-band management is efficient for high-speed interconnect networks in HPC system. However, with the development of switching chips towards a higher radix, the current implementation of the in-band management will face the problem of scalable delay performance, that is, the average switching delay of management packets increases rapidly with the increasing ports. When the number of network ports is large, the in-band management delay will increase significantly. If the problem is not solved, the technical advantages of the in-band management will disappear. Therefore, the work proposed and implemented two optimized in-band management structures, the four-quadrant double-layer ring, and the four-quadrant star ring.

The work evaluated the resource consumption and delay performance based on the Xilinx FPGA design platform and Synopsys VCS simulation platform to study the optimization of the two newly proposed in-band management structures compared to traditional ring structure. The results showed that in the high-radix switching chips with 64, 80, 96, 112, 128, 144, and 160 ports, the LUT resource occupancies of the four-quadrant double-layer ring and the star ring structures increased by an average of 5.46% and 1.71% compared to the single ring structure, respectively. Meanwhile, the occupancy of LUTRAM resource increased by an average of 30.89% and 21.81%; that of FF resource by an average of 3.86% and 0.19%; the switching delay of management packets decreased by 25.75% and 62.25%, respectively. Considering both resource consumption

and delay performance, the star ring was an ideal structure to deal with the delay performance scalability problem among the three structures, which can be applied to realize the in-band management of the high-radix switching chip in the future.

Both the double-layer ring and star ring structure proposed in the work are four-quadrant. The major factors are as follows: (1) The number of ports for most achievable higher-radix switching chips are multiples of 4. (2) If the number of quadrants is lower, such as 2 quadrants, the effect of a new structure on reducing the average switching delay of management packets may be less significant. (3) If the number of quadrants is too high, it may increase the overall wiring of the chip, which hinders the back-end design of the chip. In actual chip design, it is necessary to consider the back-end layout comprehensively and the optimization goals of delay performance to select the appropriate number of quadrants, thus balancing the demands for resource consumption, delay performance, and implementation.

References

1. Cao, J., Xiao, L., Pang, Z., et al.: The efficient in-band management for interconnect network in Tianhe-2 system. In: 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 18–26. IEEE Press, New York (2016). <https://doi.org/10.1109/PDP.2016.58>
2. Kim, J., Dally, W.J., Towles, B., et al.: Microarchitecture of a high-radix router. In: 32nd International Symposium on Computer Architecture (ISCA), pp. 420–431. IEEE Press, New York (2005). <https://doi.org/10.1109/ISCA.2005.35>
3. Kim, J., Dally, W.J., Abts, D.: Adaptive routing in high-radix Clos network. In: The 2006 ACM/IEEE Conference on Supercomputing (SC), pp. 420–431 (2006). <https://doi.org/10.1109/SC.2006.10>
4. Broadcom Corporation Homepage. <https://www.broadcom.com>
5. Centec Networks Corporation Homepage. <https://www.centecnetworks.com>
6. Mellanox Technologies Corporation Homepage. <https://www.mellanox.com>
7. Intel Corporation Homepage. <https://www.intel.com>
8. InfiniBand Trade Association: Infiniband Architecture Specification: Release 1.0. InfiniBand Trade Association (2000)
9. Birrittella, M.S., Debbage, M., Huggahalli, R., et al.: Enabling scalable high-performance systems with the intel omni-path architecture. *IEEE Micro* **36**(4), 38–47 (2016). <https://doi.org/10.1109/MM.2016.58>
10. Birrittella, M.S., Debbage, M., Huggahalli, R., et al.: Intel omni-path architecture: enabling scalable, high performance fabrics. In: IEEE Symposium on High-performance Interconnects. IEEE Press, New York (2015). <https://doi.org/10.1109/HOTI.2015.22>
11. Dally, W.J.: Virtual-channel flow control. *ACM Sigarch Comput. Archit. News* **18**(3), 60–68 (1990). <https://doi.org/10.1145/325096.325115>
12. Xilinx Corporation Homepage. <https://www.xilinx.com>
13. Synopsys Corporation Homepage. <https://www.synopsys.com>



A 32 Gb/s Low Power Little Area Re-timer with PI Based CDR in 65 nm CMOS Technology

Zhengbin Pang¹, Fangxu Lv¹(✉), Weiping Tang², Mingche Lai¹, Kaile Guo², Yuxuan Wu², Tao Liu², Miaomiao Wu², and Dechao Lu²

¹ National University of Defense Technology, Changsha, China
lvfangxu1988@163.com

² Air Force Engineering University, Xi'an, China

Abstract. This paper presents a 32 Gb/s low power little area re-timer with Phase Interpolator (PI) based Clock and Data Recovery (CDR). To further ensure signal integrity, both a Continuous Time Linear Equalizer (CTLE) and Feed Forward Equalizer (FFE) are adapted. To save power dissipation, a quarter-rate based 3-tap FFE is proposed. To reduce the chip area, a Band-Band Phase Discriminator (BBPD) based PI CDR is employed. In addition, a 2-order digital filter is adopted to improve the jitter performance in the CDR loop. This re-timer is achieved in 65 nm CMOS technology and supplied with 1.1 V. The simulation results show that the proposed re-timer can work at 32 Gb/s and consumes 91 mW. And it can equalize > -12 dB channel attenuation, tolerate the frequency difference of 200 ppm.

Keywords: Re-timer · Clock and Data Recovery (CDR) · Phase Interpolator (PI) · Feed Forward Equalizer (FFE)

1 Introduction

The continuously increasing bandwidth demand for data communication in high performance computer (HPC) has pushed wire-line connections towards data-rates of 25 Gb/s or beyond [1]. However, low-power and high density data transceivers are also key elements of modern HPC, due to systems such as network switches and processor interfaces will employ optical communication [2, 3]. Figure 1 shows the next switch system with optical communication. The black box in the center of the system, which outputs optical signal directly, usually consists a switch chip, many re-timer chips and other optical chips. However the bandwidth, power efficiency and area of the re-timer also limit performance of the switch system. Even though, many reported CDR can meet its bandwidth, but their power is hungry due to fabricated with III-VI materials [4]. In addition, the large area of the CDR is not good for high density integrated.

To solve these problems, a high speed, low power and little area re-timer based CMOS technology is proposed. To save the power dissipation, a quarter-rate based 3-tap FFE is proposed. To reduce the chip area, a BBPD based PI CDR is employed. In addition, to improve the high speed performance, a 2-order digital filter is used.

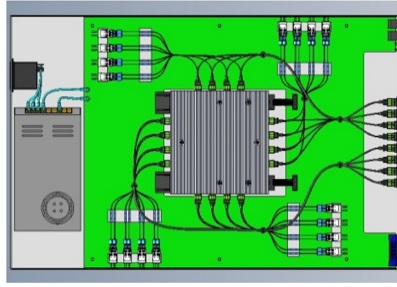


Fig. 1. System of the next switch chip with optical network.

This paper is organized as follows. Section 2 presents the architecture of the re-timer, followed by the description of building blocks. Section 3 reveals the experimental results and the conclusion.

2 Architecture and Circuit Design

Figure 2 shows the re-timer architecture, which includes a phase tracing control loop and a data path. In the phase tracing control loop, the input data are sampled by 1/4 rate 8 phase clocks firstly. Secondly, the early/late information between the sampling clocks and input data is extracted by PD circuit. After the voter and filter, the control words generated by code circuit are used to rotate the extra input clock to match the phase of the input data. In the data path, firstly, the input data is equalized by the CTLE. Secondly, it is resampled by the recovery clock. Lastly the data is equalized and output by the 3-tap FFE with driver.

In the phase tracing control loop, a quarter rate BBPD based CDR is introduced, which consists of 1/4 rate sampler, 8:32 DEMUX, phase detector, voter, 2-order digital filter, code, and phase interpolator. The data path consists CTLE, baud rate sampler, delay latch array, 4:1 MUX based 3-tap FFE.

2.1 BBPD Based PI-CDR with 2-Order Digital Filter

Clock recovery circuit is the most important circuit module in high re-timer system. Its main task is to extract clock information from the input data with amplitude noise and phase noise, and then retiming the data. In addition, CDR can track the low frequency phase jitter introduced in the input data. The working principle of a CDR, shown in Fig. 3, mainly includes clock recovery (CR) module and data recovery (DR) module. The CR detects the phase information of the data, and then generates the clock related to the input data. The DR uses the generated clock to complete the data retiming task.

Figure 4 shows the model of the proposed CDR, which is a Bang-Bang phase discriminator (BBPD) based PI CDR with 2-order digital filter. It consists a BBPD, a voter, a 2-order digital filter, a phase interpolator, and a feed-back. The BBPD is used to extract the phase error between the input data and clock generated from PI. The voter is used to get the efficient results of the decision from BBPD. The 2-order digital filter is adapted

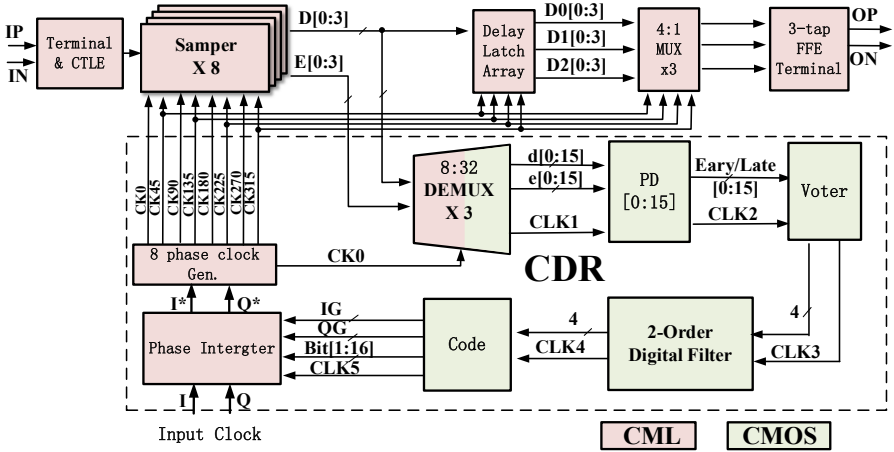


Fig. 2. Re-timer architecture.

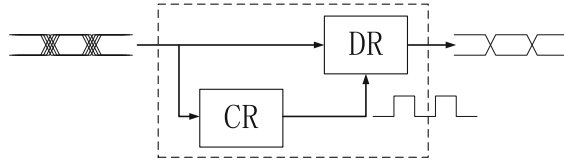


Fig. 3. Basic working principle of CDR.

to smooth the result of the voter and then used for PI. PI is used to generate a desired phase clock with a fixed input clock (Fig. 5).

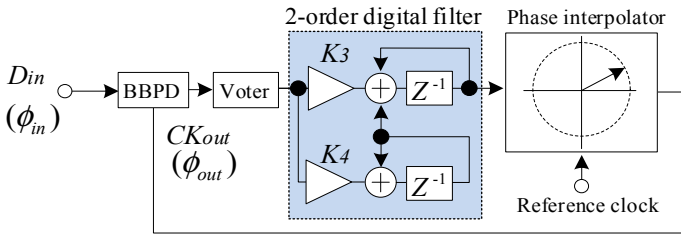


Fig. 4. BBPD based PI CDR with 2-order digital filter.

To analyze the performance of the CDR, a linearized model with parameters is modeled in Fig. 6. In the linearized model, K_{TD} is the edge conversion density of the input data. K_{PD} is the phase detector gain. K_V is the gain of the voter to take effects of decimation from any decimation that takes place. The value K_P and K_I correspond to the proportional and integral paths from the output of the voting to the PI. K_{PI} is the gain of the PI. This corresponds to the resolution of the PI in units of Unit Interval (UI) per bit. z^{-NEL} represents all of the delay (analog and digital pipe stages) in going around the

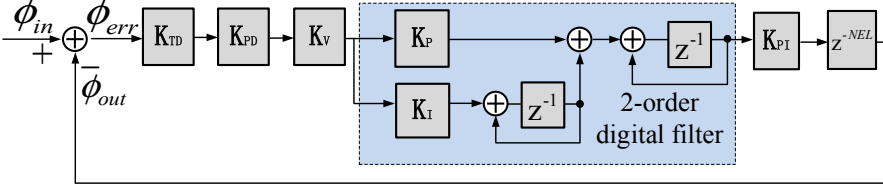


Fig. 5. Linearized model of the CDR.

loop. Thus, the open-loop transfer function for the linearized CDR can be express as

$$G(Z^{-1}) = \frac{\varphi_{out}}{\varphi_{err}} = K_{TD}K_{PD}K_V \left(K_P + K_I \frac{Z^{-1}}{1 - Z^{-1}} \right) \frac{Z^{-1}}{1 - Z^{-1}} K_{PI} Z^{-NEL} \quad (1)$$

In Z-Domain, $z = e^{S * T_{DLF}}$, where S is $j * 2\pi f$ and T_{DLF} is the operation (cycle) period of digital loop filter (DLF). In addition, $e^{-sT_{DLF}} = 1 + (-sT_{DLF}) + \frac{(-sT_{DLF})^2}{2} + \frac{(-sT_{DLF})^3}{3!} + \dots$, when $sT_{DLF} \ll 1$, we can get

$$z^{-1} = e^{-sT_{DLF}} \approx 1 - sT_{DLF}, (sT_{DLF} \ll 1) \quad (2)$$

Therefore the open-loop transfer function can be given by

$$\begin{aligned} G(S) &= K_{TD}K_{PD}K_V \left[\frac{K_P(1 - sT_{DLF})}{sT_{DLF}} + \frac{K_I(1 - sT_{DLF})^2}{s^2T_{DLF}^2} \right] K_{PI}(1 - sT_{DLF})^{NEL} \\ &\approx K_{TD}K_{PD}K_V \left(\frac{K_P}{sT_{DLF}} + \frac{K_I}{s^2T_{DLF}^2} \right) K_{PI}(1 - sT_{DLF})^{NEL} \end{aligned} \quad (3)$$

The phase transfer function is given by the following well known equation:

$$H(S) = \frac{\varphi_{out}}{\varphi_{in}} = \frac{G(S)}{1 + G(S)} \quad (4)$$

Figure 6 shows the calculated phase transfer function of the proposed CDR. It can be observed that the bandwidth is 1.46 MHz.

2.2 Phase Interpolator

PI is the key module in the CDR. It can generate a desired phase clock underling the control of the input control words for sampling the input data. The working principle of the basic PI can be explained by a vector diagram and its mathematical model equation. In Fig. 7, the two basic vectors \vec{V}_Q and \vec{V}_I , which between the phase is 90° , can composite a new vector. It's known by the vector knowledge of geometry that, the phase of the composite vector, which is the angle between the new composite vector and the horizontal vector, can be controlled through changing these amplitudes of the two basic vectors. And the geometry theory of this composite vector can be expressed by Eq. (5).

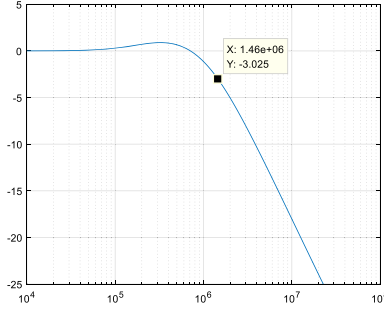


Fig. 6. Calculated the transfer function of the proposed CDR.

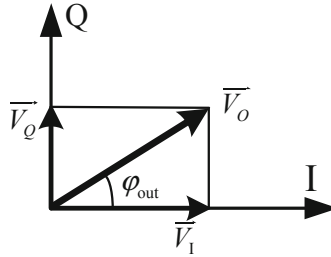


Fig. 7. Composite vector.

$$\vec{V}_O = \vec{V}_Q + \vec{V}_I \quad (5)$$

$$V_{out} = \alpha A \sin(\omega t) + (1 - \alpha)A \cos(\omega t), \quad (0 \leq \alpha \leq 1) \quad (6)$$

$$V_{out} = A\sqrt{\alpha^2 + (1 - \alpha)^2} \sin(\omega t + \varphi_{out}) \quad (7)$$

$$\varphi_{out} = \arctan\left(\frac{1 - \alpha}{\alpha}\right) \quad (8)$$

In actual circuit, the two basic vectors \vec{V}_Q and \vec{V}_I can be replaced by $\alpha A \sin(\omega t)$ and $(1 - \alpha)A \cos(\omega t)$, thus Eq. (5) can be expressed as Eq. (6), which of the value is limited in $[0, 1]$. The phase between $\sin(\omega t)$ and $\cos(\omega t)$ is 90° , αA and $(1 - \alpha)A$ are their amplitudes respectively. When α is changed, the phase of the V_{out} followed in 0 to 90° , which is the desired phase of V_{out} . In order to precisely calculate the output phase, the Eq. (7) can be derived by Eq. (6), and the phase of V_{out} can be calculated by Eq. (8). Figure 8 shows the V_{out} waveforms with different α values.

Figure 9 shows the part circuit of the PI. It includes two pull-up loads, two pairs of input transistors, and 16 equivalent tail current sources under each of input pairs. And the relationship between input temperature code and output clock phase is depicted in Fig. 10.

If the input two basic clocks are changed from $0, 90, 180, 270$, the phase of the composited clock can be got in any degree ($0-360$) that we are desired, which is depicted

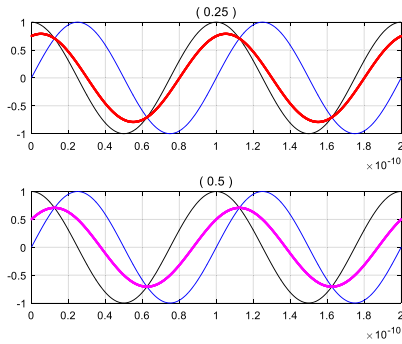


Fig. 8. Different output clocks with different α value.

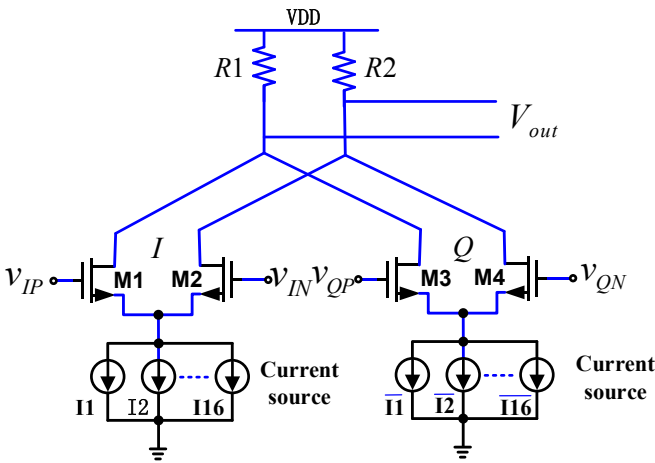


Fig. 9. Part circuit of the PI.

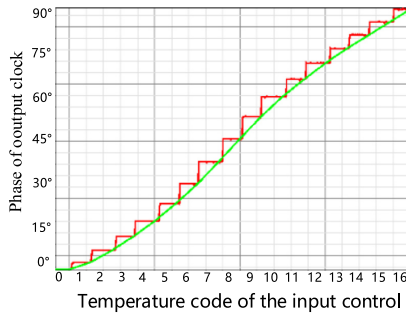


Fig. 10. The relationship between input temperature code and output clock phase.

in Fig. 11. Figure 12 shows the circuit of the complete PI, which consists of 4 pairs of the input transistors, control words transistors and tail current sources.

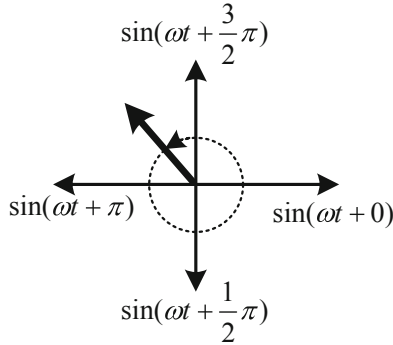


Fig. 11. 360° output phase of the composite vector.

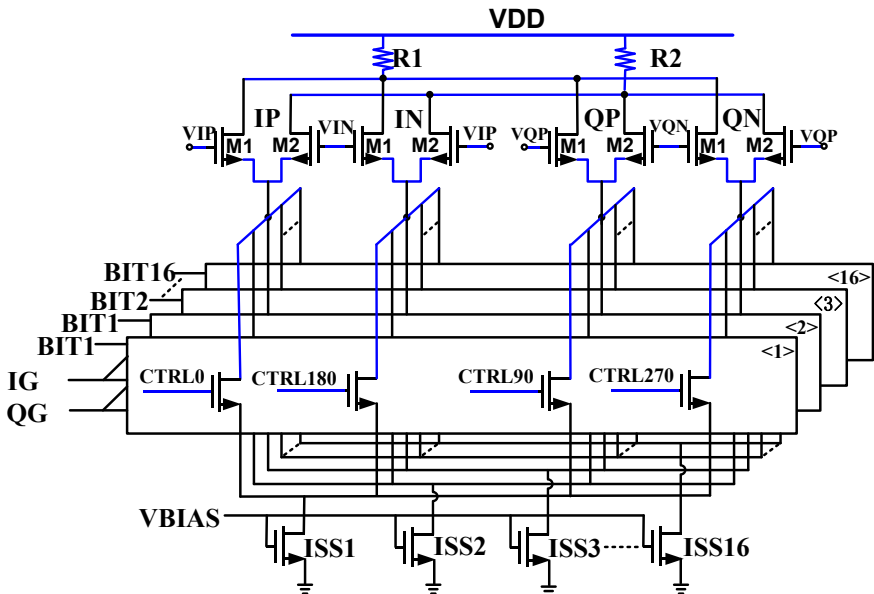


Fig. 12. Circuit of the complete PI.

2.3 4:1 MUX Based 3-Tap FFE

As everyone knows that, the dielectric channel usually presents low - pass characteristics due to the dielectric loss and skin effect. Figure 13(a) shows a typical backbone channel S12 curve, which includes a 19 in. PCB channel, 2 via holes, 2 packages and 2 connectors. The attention at the baud rate frequency is -17.32 dB. When data rate exceeds the channel bandwidth, the high data rate signal couldn't transform within 1 unit interval (UI) and extend to the adjacent signal interval, which are showed in Fig. 13 (b), and this phenomenon is usually called inter-symbol interference (ISI). ISI can deteriorate signal

integrity of the high speed signal. Figure 14 presents a 32 Gb/s NRZ eye diagram before this channel, and the eye diagram after passing channel is closed due to the ISI.

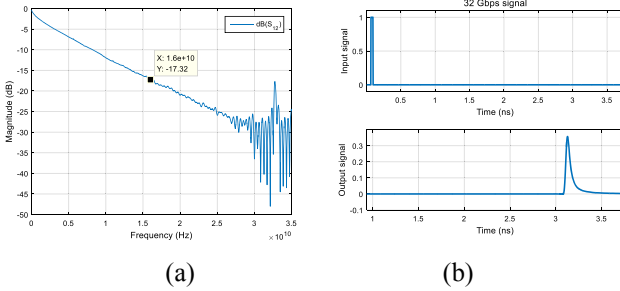


Fig. 13. (a) S12 curve of a typical channel, (b) unit pulse response before and after the channel.

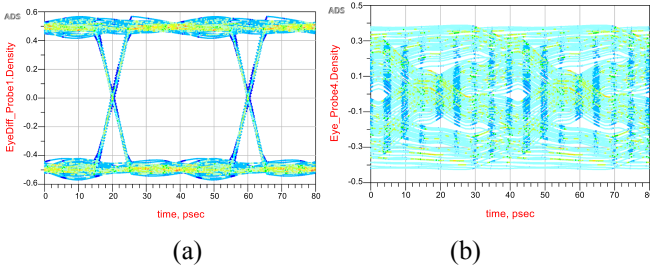


Fig. 14. (a) Eye diagram before the channel, (b) eye diagram after the channel.

In order to mitigate this problem, a feed-forward equalizer (FFE) is usually to be introduced at the output of the re-timer to reduce the ISI. The basic construction of a 3-taps FFE as show in Fig. 15, which includes 3 delay units, 3 multiplying units with 3 coefficients and a summer, is a finite impulse response (FIR) filter. The time-domain transfer function is Eq. (9), and the Z-domain transfer function is Eq. (10), where the Z is $e^{j2\pi fT}$. Figure 16 shows the channel response with different character. The black curve presents the channel response without FFE. The blue curve describes a high pass based FIR filter with proper 3 tap coefficients. And the red curve shows the channel response with the FFE, which can keep the signal integrity. Figure 17 (a) and (b) show the eye diagrams before and after the channel with proper coefficients based FFE.

$$y(t) = c0 * x(t) + c1 * x(t + T) + c2 * x(t + 2T) \tag{9}$$

$$H(Z) = c0 * Z^0 + c1 * Z^{-1} + c2 * Z^{-2} \left(z = e^{j2\pi fT} \right) \tag{10}$$

Compared with the pre-emphasis based FFE, the de-emphasis based FFE is widely used due to its simple circuit structure. A de-emphasis based FFE equalizes the output's signal through reducing the amplitude of the high frequency components of the original

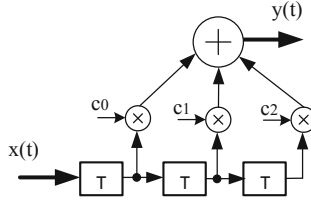


Fig. 15. Basic construction of FFE.

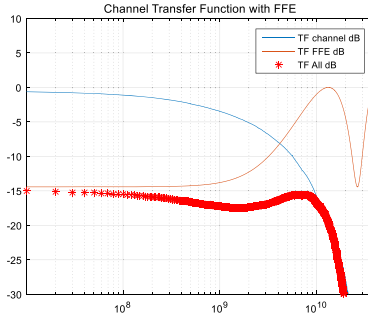


Fig. 16. Frequency domain channel response.

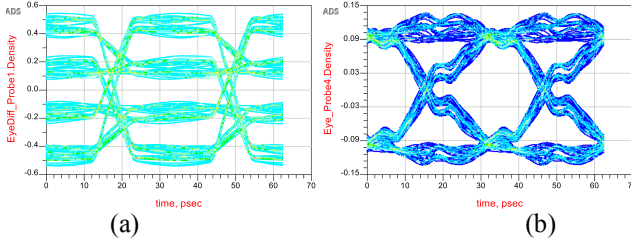


Fig. 17. Time domain channel response (a) eye diagram before channel, (b) eye diagram after channel with proper coefficients FIR.

signal and maintaining the amplitude of the low frequency components of the original signal, which still follows the principle of the FFE. However, when data rates exceed 20 Gb/s, the high speed delay is power hungry and the timing is constrict under PVT variation. In order to solve these problems, a 4:1 MUX based 3-tap FFE is introduced to this re-timer showing Fig. 18. Compared with other FFE circuits, the delay cell in this FFE circuit designed with 3 4:1 MUX units, which can save power and relaxes the critical path timing by using the quarter rate clock and avoiding CML based circuits.

Figure 19 describes the 4:1 MUX with its timing diagram. This MUX consists of shunt-peaked loads and four identical unit cells, which is activated sequentially by the 2UI-spaced pulses quadrature clock (i.e., CK0, CK90, CK180, and CK270) to combine the four quarter-rate data into one serial sequence.

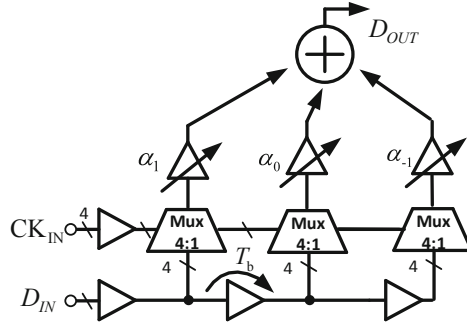


Fig. 18. Multiple-MUX based FFE.

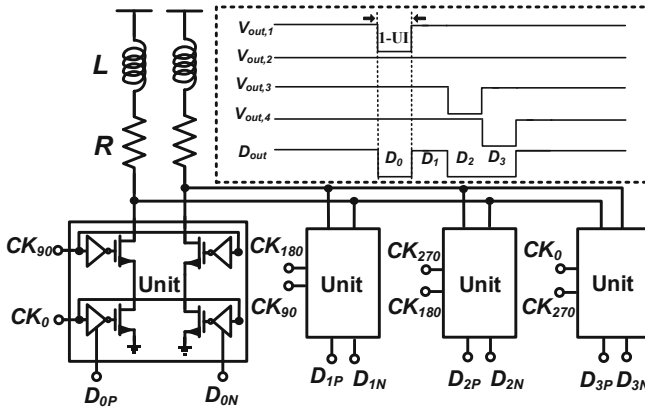


Fig. 19. The 4:1 MUX with its timing diagram.

3 Experimental Results

The re-timer designed in 65 nm CMOS Technology. The layout of the re-timer is shown in Fig. 20. The core area of this chip is 0.11 mm².

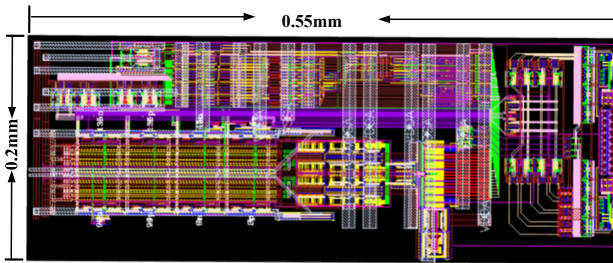


Fig. 20. Layout of the re-timer.

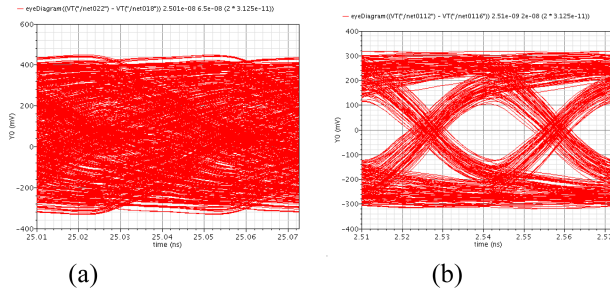


Fig. 21. Eye diagram of equalized signal (a) without FFE, (b) with proper coefficients of FFE.

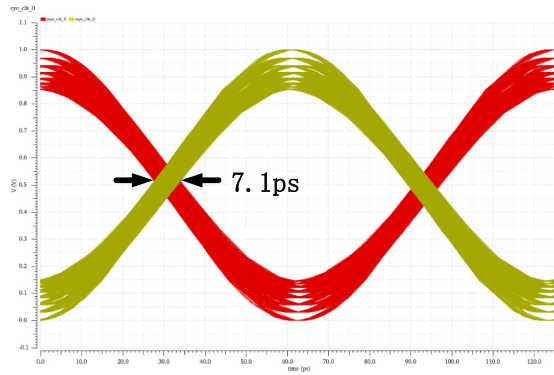


Fig. 22. Eye diagram of the recovery 1/4 rate clock with 200 ppm frequency difference.

Figure 21 show the 32 Gb/s output eye diagram of this re-timer with or without FFE. When it passes a -12.52 dB@16 GHz attenuation channel without FFE, the output eye-diagram is closed as shown in Fig. 21 (a). When using the 3-tap FFE with the proper coefficients, the vertical eye opening of the eye diagram is 200 mVpp just as shown in Fig. 21(b). When setting 200 ppm frequency between the input data and the reference clock, the eye diagram of the recovery 1/4 rate clock is shown in Fig. 22, and the total jitter of that is 7.1 ps. The total power of this re-timer is 91 mW under 1.1 V supply. Table 1 compares the performance of this work with prior similar works.

Table 1. Performance summary

	This work (Simulation)	5 (Fabricated)	6 (Fabricated)
Data rate	32 Gb/s	32 Gb/s	26.5 Gb/s
Power	91 mW	102 mW	254 mW
CDR technology	PI and 2-order digital filter	DCO and digital filter	LC-QDCO and digital filter
Recovered clock Jitter (PP)	<7.1 ps	N/A	<8.9 ps
CoreArea (mm ²)	0.55 × 0.2	0.8 × 0.28	1 × 0.75
Technology	65 nm	28 nm	65 nm

4 Conclusion

In order to solve the problem of high power consumption and large area of the high speed re-timer in HPC data communication, a 32 Gb/s low power little area re-timer with PI based CDR is proposed. To further ensure signal integrity, both a CTLE and feed forward equalizer are adapted. To save power dissipation, a quarter-rate based 3-tap FFE is proposed. To reduce chip area, a BBPD based PI CDR is employed. In addition, a 2-order digital filter is adopted to improve the high speed performance in the CDR loop. This re-timer is achieved in 65 nm CMOS technology and supplied with 1.1 V. The simulation results show that the proposed re-timer can work at 32 Gb/s and consumes 91mW. The 3-tap FFE in the re-timer can equalize > -12 dB channel attenuation. The PI based CDR with 2-order digital filter can CDR can tolerate a frequency difference of 200 ppm.

References

1. Rupp, K.: 42 years of microprocessor trend data. <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>
2. Moore, G.E.: Cramming more components onto integrated circuits. *Electronics* **38**(8), 114–117 (1965)
3. Pham, D.: The design and implementation of a first-generation CELL processor-a multi-core SoC. In: 2005 International Conference on Integrated Circuit Design and Technology, Austin, TX, USA, pp. 49–52. IEEE (2005)
4. Nagashima, K.: 28-Gb/s × 24-channel CDR-integrated VCSEL-based transceiver module for high-density optical interconnects. In: 2016 Optical Fiber Communications Conference and Exhibition (OFC), Anaheim, CA, pp. 1–3. IEEE (2016)
5. Rahman, W.: A 22.5-to-32-Gb/s 3.2-pJ/b Referenceless Baud-Rate Digital CDR With DFE and CTLE in 28-nm CMOS. *IEEE J. Solid-State Circ.* **52**(12), 3517–3531 (2017)
6. Chu, S.-H.: A 22 to 26.5 Gb/s optical receiver with all-digital clock and data recovery in a 65 nm CMOS process. *IEEE J. Solid-State Circ.* **50**(11), 2603–2612 (2015)



DBM: A Dimension-Bubble-Based Multicast Routing Algorithm for 2D Mesh Network-on-Chips

Canwen Xiao, Hui Lou, Cunlu Li^(✉), and Kang Jin

College of Computer, National University of Defense Technology,
Changsha 410073, Hunan, China
{cw-xiao, huilou, cunluli, kangjin}@nudt.edu.cn

Abstract. Network-on-Chips (NoCs) has been widely used today for efficient communication in multicore systems. Existing NoCs mostly use 2D mesh topology in commercial and experimental manycore processors since it maps well to the 2D layout. For 2D mesh, dimension order routing and different adaptive routing algorithms perform well in unicast traffic but suffer from poor performance when faced with one-to-many (multicast) traffic. Efficient multicast routing algorithm is an important target for the design of special on-chip networks such as neural networks. Recently proposed multicast routing algorithms are less efficient or can introduce unbalanced load in some situations. In this paper, we propose DBM, a novel multicast routing algorithm based on the dimension-bubble flow control for 2D mesh networks. DBM is deadlock-free while achieving the minimal path and fully-adaptive multicast routing algorithm. Moreover, DBM simplifies the deadlock condition where the escape channel is not necessary. Evaluation results show that DBM can achieve much better performance than existing multicast routing algorithms, with 18% reduction in packet latency and 16% improvement in network throughput.

Keywords: Dimension-bubble flow control · Multicast routing algorithm · Deadlock

1 Introduction

Network-on-Chips (NoCs) has always been a challenging research topic, providing a scalable solution for Multiprocessor System-on-Chip (MPSoC). 2D mesh topology is usually preferred due to its layout on a planar surface in the chip. The topology of a 4-ary 2-cube mesh and corresponding router microarchitecture are presented in Fig. 1.

In addition to unicast communication, NoCs also need to deal with a lot of multicast communication [3]. Multicast messages are useful for efficient execution of parallel programs as the multicast communication is frequently employed in many MPSoC applications such as replication [8], barrier synchronization [13],

cache coherency in distributed shared-memory architectures [6] and clock synchronization [1]. In these MPSoC applications, it is a key issue to ensure efficient communication for multicast packets. On the other hand, the number of processor cores integrated on the chip is also increasing. For example, SpiNNaker project aims to produce 10,000-core chips for modeling of large-scale spiking neural networks in biological real time [9]. For these million processor machines, multicast packets with appropriate multicast routing algorithms can effectively reduce the number of packets in the network to alleviate network congestion.

Some theories and methodologies have been proposed [4, 7, 10, 11] to achieve deadlock-free multicast routing. Virtual circuit tree multicasting (VCTM) [4], as a representative, achieves a tree-based routing algorithm to support multicasting in NoCs. VCTM builds several virtual circuit trees through the destinations before the multicast messages are injected into the network. VCTM achieves this scheme by sending separate unicast setup messages (look ahead signals) for each destination, through the utilization of virtual circuit table (VCT) and content addressable memory.

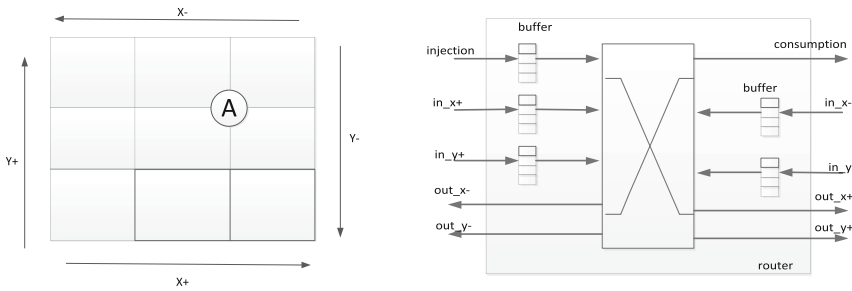


Fig. 1. 4-ary 2-cube mesh

In VCTM, cyclic dependencies can be avoided by using the Dimension Order Routing (DOR) algorithm for both the setup and the multicast messages. However, some shortcomings can be introduced within VCTM. First, VCTM's design complexity and hardware overhead strongly depends on the network size, making it difficult to scale up. Second, VCTM is less efficient when faced with high injection rate network conditions. Third, when updating the VCT, the source node has to send discrete unicast setup messages per destination. In this situation, when faced with large number of destinations, the number of unicast setup message will be increased, thereby reducing the performance. Recursive Partitioning Multicast (RPM) [11] is another representative multicast routing algorithm. In RPM method, the processing of the header information is complex and will be performed several times for each multicast message. VCTM and RPM share the same disadvantage that a message may hold several output channels, thereby increasing network contention. Finally, both RPM and VCTM are based on deterministic algorithms and cannot provide adaptiveness to neither unicast

nor multicast messages. Paper [7] presents a routing algorithm called Balanced Adaptive Multicast (BAM). This algorithm adopted Duato's principle [2] to realize the deadlock-free adaptive routing.

In our former works, the Dimensional Bubble Routing Algorithm (DBRA) [12] is proposed for Mesh networks. This algorithm realizes a fully adaptive routing for unicast communication without the escape channels. In this paper, DBM, a novel dimension-bubble-based multicast routing algorithm is studied based on the idea of DBRA. The contributions of this paper are as follows:

1. The strategy of dimensional-bubble flow control is presented for multicasting operation in 2D Mesh networks and the novel multicast routing algorithm, DBM, is studied;
2. We proof and present that DBM is deadlock-free with efficient multicast communication;
3. We provide a thorough evaluation of the proposed organization and demonstrate that we can achieve higher performance.

The rest of this paper is organized as follows. In Sect. 2, the novel multicast routing algorithm will be presented. In Sect. 3, we prove that the proposed flow control strategy can ensure that the minimal path and fully-adaptive routing algorithm is deadlock-free. The performance of the novel algorithm is evaluated in the Sect. 4. In the end, we summary this paper in Sect. 5.

2 Novel Multicast Routing Algorithm

In this section, we design DBM, a novel multicast routing algorithm based on the dimensional-bubble flow control. Firstly, the algorithm schemes of RPM and BAM will be analyzed and the novel multicast routing algorithm will be presented based on the study of RPM and BAM.

2.1 RPM and BAM Routing Algorithm

RPM algorithm uses the determinate method to divide the network to eight regions according to the router's location. Then, according to the destinations of the multicast packet, the output port of packet will be calculated by deterministic rule. RPM algorithm ensures the multicast packets are transmitted along the same path as more as possible. At the same time, RPM also strives to balance the load of network. Figure 2 depicted an example of eight regions of RPM in 4-ary 2-cube Mesh network.

RPM algorithm adopts two virtual networks called VN0 and VN1 to avoid the existence of deadlock routing in the network. However, this method can bring unbalanced network communication to degrade the performance. Similarly, BAM algorithm also divides the network to eight regions depended on the location of the multicast packet. BAM multicast routing algorithm is based on the strategy of full-adaptive routing of Duato's principle, and choose the output port with lower buffer utilization when there exist two or more available output ports.

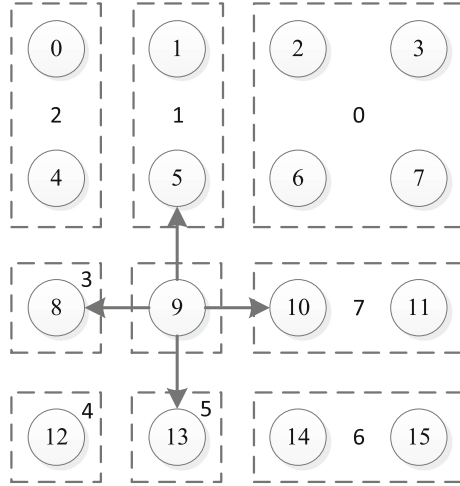


Fig. 2. An example of eight regions of RPM

2.2 Dimension-Bubble Multicast (DBM) Algorithm

We propose the novel multicast algorithm called Dimension-Bubble Multicast (DBM) based on the study of RPM and BAM algorithm. At the same time, DBM algorithm adopts the strategy of minimal path and realizes the multicast routing based on the idea of DBRA algorithm [12].

In DBRA, the definition of dimensionbubble flow control for unicast communication is as follows:

when a packet wants to move to the next buffer, if there are remaining routing hops in N dimensional directions ($N \leq n$), then this packet can request for arbitration of the next buffer only when there are more than or equal to N free packet spaces in the next buffer. Otherwise, it has to wait.

DBRA routing algorithm uses the remaining number of hops in a dimension to judge the next step of packet’s routing. In order to support the multicast routing, we propose a new strategy of flow control based on DBRA in 2-dimensional (2D) Mesh networks.

In 2D Mesh networks, we define the set of destination nodes of a multicast packet as

$$\{D_1, D_2, \dots, D_i, \dots, D_n | 1 \leq i \leq n, n \text{ is an integer}\}$$

Suppose that the packet needs to transmit M_i dimensional directions before arriving at the destination node D_i . We define $\text{Max}\{M_i\}$ to represent the maximum value in the set of $\{M_1, M_2, \dots, M_i, \dots, M_n | 1 \leq i \leq n, n \text{ is an integer}\}$.

Based on the above definitions, the novel flow control strategy can be described as follows:

The multicast packet can request for arbitration of the next buffer only when there exist more than or equal to $\text{Max}\{M_i\}$ free packet space in the input buffer of the next-hop router. Otherwise, it has to wait.

We name the new flow control strategy as DBMFC (Dimensional Bubble Multicast flow control). For 2D Mesh networks, once a multicast packet arrived at the input buffer of a router, it may have remaining routing paths in $X+/X-$, $Y+/Y-$ dimensional directions. But if only generic minimal path can be chosen in 2D Mesh networks, each destination node has routing hops only in two dimensions at most.

Therefore, $\text{Max}\{M_i\} \leq 2$ can be guaranteed in 2D Mesh networks, and thus it is enough for the input buffer to be set with 2-packet size, which can meet the demand of DBMFC for multicast operations.

Based on DBMFC, we propose DBM, to achieve a fully-adaptive multicast routing algorithm. The following is the description of DBM algorithm:

- Firstly, minimal-path routing is adopted as the baseline in 2D mesh networks. For each multicast packet in the input buffer, it is calculated how many hops this multicast packet must transmit on the different dimensions for each destination node.
- Secondly, multicast packets calculate the remaining dimensions of the destination nodes and appeal the arbitration requests of the different buffers meet with the strategy of DBMFC.
- Thirdly, if the number of granted requests is more than one, DBM algorithm will choose an output port with lower buffer utilization in the next-hop router. This process is similar to RPM and BAM algorithms.
- Fourthly, the replicated packet carried the information of those destination nodes had remaining hops in the dimensional direction to flow out. At the same time, the number of hop of destination node in this dimensional direction minus 1.
- The multicast packets repeat from step 1 to step 4 until all the destination nodes have been traversed.

Compared with DBRA algorithm for unicast routing, DBM algorithm decides the next routing dimensional direction by the value of $\text{Max}\{M_i\}$. The choosing strategy of arbitration of DBM algorithm is similar as the arbitration strategy of RPM and BAM based on regions and the entire network is divided into eight regions labelled as 0, 1, 2, 3, 4, 5, 6 and 7 such as Fig. 2.

We explain the routing strategy of DBM algorithm in Fig. 2. Suppose that the multicast packet hops on the $Y+$ direction. When all destination nodes of packet locate in the region $Y+$ (region 1 in Fig. 2), it means that the set of destination nodes do not include those nodes in region 0 and 2.

If there are free spaces in the next router of $Y+$ direction, the packet may enter to the buffer of the next router. On the other hand, if there are nodes of region 0 or region 2 in the destinations of packet, the packet may enter when there are more than or equal to 2 free packet spaces of the next router in $Y+$ direction.

For DBM routing algorithm, asynchronous replication is adopted. In asynchronous replication, branch replicas will not block each other, since each of them proceeds independently. It means that replicated packets can be granted in different directions respectively.

3 Proof of Deadlock Freedom of DBM

The goal of this section is to explain how the DBM algorithm achieves the dead-lock free characteristic for any minimal path, adaptive routing on 2D mesh networks.

DBM algorithm is achieved based on DBRA algorithm which is fully adaptive routing for unicast operations. Since the deadlock freedom of DBRA has been proved in paper [12], if the differences between DBM and DBRA do not cause dead lock of network, it can be concluded that DBM algorithm is deadlock freedom.

The differences between DBRA and DBM are that multicast packets may carry with more than one addresses of destination nodes and the replication operation of packet will be performed in the router and it will increase the number of packets in the network. We should analyze whether the replication operation will cause dead lock in the network.

Proof Sketch: In each step of the proof, we will analyze all possible cases of packet in the network and present allocation of buffers to prove that all kinds of packets can reach the destination node. Accordingly, the conclusion that the deadlock does not exist in the network can be made as a result.

Proof. If the replication operation was committed before multicast packets injecting into the network, this replication operation cannot cause the dead lock in the network. This is because the replicated packets will be injected into the network as same as the unicast packets,

Next, we analyze the replication operation after multicast packets have been injected into the network. In this situation, the multicast packets may be stored in the buffer of one-dimensional direction such as $X+$, $X-$, $Y+$ and $Y-$ in the 2D mesh networks. Without loss of generality, we assume multicast packet is in the buffer of direction $X+$. The multicast packets on buffer space have three cases:

- The first case is that remaining hops of all destination nodes of multicast packet are in the direction $X+$, it means that the value of $\text{Max}\{M_i\}$ of this packet is 1.
 Suppose: the space of buffer can contain two packets. The situation of packet is described by the third graph.

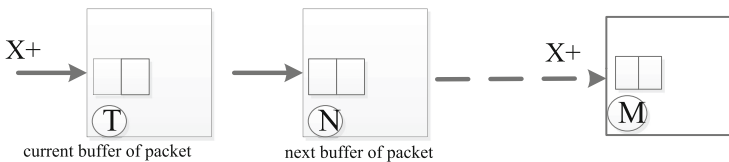


Fig. 3. Buffers in $X+$

There exist two possible cases for the next buffer. If there is one free space in the next buffer, according to the flow control strategy of DBMFC, since the value of $\text{Max}\{M_i\}$ is equal to 1, the replicated packets can enter the next buffer. If there is not room in the next buffer, we can conclude that there is no less than one packet in the next buffer whose destination nodes are only in $X+$ or packet is waiting for consumption in the next buffer. Suppose that the destination nodes of two packets in the next buffer remain hops in two directions or one direction that is not $X+$, According to the flow control strategy of DBMFC, when $\text{Max}\{M_i\}$ is equal to 2, it is necessary that the destination buffer must own two free packets space at least. So, it is impossible that the destination nodes of two packets in the next buffer remain hops in two directions or one direction that is not $X+$. We can reach the conclusion that there is no less than one packet in the next buffer whose destination nodes are only in $X+$ or packet s waiting for consumption in the next buffer. The situation of the latter buffer is the same as the next buffer.

Because there are not wraparound connections in X and Y direction, the cyclic dependency cannot be formed in X or Y direction. As a consequence, the forever block is not formed between packets of this case. Because it is impossible that the destination nodes of packets in the last buffer in $X+$ direction need to hop in $X+$ direction, it is true that packet waiting for consumption exist in the last buffer. The packet o will be consumed soon, thus, these replicated packets of the first case can always move and reach the destination node.

- The second case is that it is existed that the destination nods of multicast packet in buffer have only one-direction routing and different direction with the buffer space where the packet is placed. We analyze the possibility of packet of destination's buffer when there is not room in the destination's buffer. Suppose that the packets that are occupying the destination's buffer space are the packets waiting for consumption or the packets of the first case. Because the packets of two cases can always go ahead from the current buffer, they will not block the packets of the second case forever. Suppose that the packets that are occupying the destination's buffer space are those packets whose destination nodes remain X and Y direction routing or are also packets of the second case. According to DBMFC, these packets cannot occupy all buffer space and the destination buffering must remain one free packet space after they enter the destination buffer. Thus, the replicated packets of the second case can enter the destination's buffer space. So, the packets of the second case can always reach the destination nodes.
- The last case is that it is existed that the destination nods of multicast packet in buffer remains X and Y direction routing. According to DBM algorithm, the replicated packet remaining X and Y direction routing can have requests in X and Y direction at the same time. We consider the situation of those buffers which have the same direction with the buffer in which the packet is placed at present. Suppose the direction of current buffer space is $X+$. The situation of buffer is the same as the Fig. 3.

According to the above analyses, if the packets that are occupying the next buffer space are those packets of the above two cases or waiting for consumption, they cannot result in other packets blocking forever. As a result, the deadlock maybe only exists among the packets of the last case. According to DBM algorithm, it is certain that those packets of the last buffer in $X+$ direction have one-direction routing at most, because they have the routing in $X+$ direction no longer. These packets are the packets of the former two cases or waiting for consumption. Because the packets in the buffer space in $X+$ direction can move certainly, the packets of the last case may finish the routing of one-direction and become the packets of the former two cases or waiting for consumption. So, the packets of the last case will also reach the destination node.

Based on the above proof, we can conclude that DBM will not introduce deadlock that when the scheme is deadlock free in the 2D mesh.

4 Evaluation

In this section, we study performance and scalability of DBM algorithm supporting fully-adaptive multicast routing. The DBM algorithm is common minimal path, fully-adaptive routing algorithm except for DBMFC. We base our evaluation on the BookSim simulator [5] developed at the Stanford University, thanks to its modular design, and the availability of a large variety of classic network implementations. DBM algorithm was implemented in the BookSim simulator with little effort.

We compare average packet latencies of DBM to its counterparts: RPM and BAM algorithms. More specifically, for BAM algorithm, Duato’s method reserves one virtual channel, which employs dimension order routing (DOR), as an escape channel, and the other virtual channels employ the shortest path adaptive routing algorithm.

4.1 Experiment Setup

It is noteworthy to mention that in our design, DBM takes the same way as DBRA to allow a packet to be chosen for the arbitration independently, without considering its position in the buffer. In other words, head of line (HOL) blocking will not occur in DBM. To ensure fair comparisons, we have removed HOL blocking from the implementations of RPM and BAM, which also stresses the effect of multicast routing strategies. In addition, we have assigned the same amount of buffer space to each router in the evaluation, and in the case of BAM, uses exactly the same adaptive routing function as DBM.

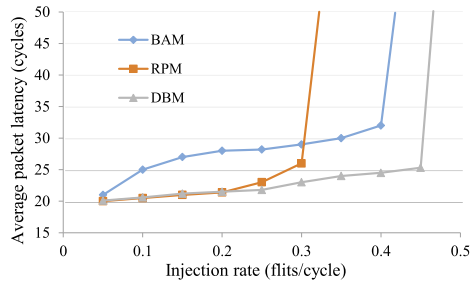
The simulator is warmed up for 10,000 cycles and then the performance is measured over another 100,000 cycles. Network is considered unstable when average latency of one packet exceeds 1000 cycles [5], therefore we stop reporting results when latency is beyond this point. Flit injection rate used in the simulation is defined as the time to take a single flit to be injected at a source. For example, injection rate is 0.1 means that each source injects a new flit in one out of every ten simulator cycles. Our simulation settings are summarized in Table 1.

Table 1. BookSim configuration parameters

Configuration parameters	Value
Topology	4-ary 2-cube mesh, 8-ary 2-cube mesh
Number of VCs	4, 6, 8
Proportion of multicast packets	10%, 15%, 20%
Injection rate (flits/cycle/node)	0.1–0.5
Number of destinations in multicast packets	6, 9, 12, 15
Traffic pattern	Uniform random, random permutation, bit rotation, transpose

4.2 Average Latency and Load Scalability

The Fig. 4, 5, 6, 7 plots the average latencies for 2-cube mesh network under different patterns. DBM outperforms the other two algorithms in all cases. In particular, under the pattern of uniform random, the saturation throughput of DBM algorithm is largest in three algorithms and BAM is better than RPM algorithm. When the load of network is low, the latency of three algorithms is almost same. However, with the increase of load of the network, the performance of RPM is the worst. It shows that the load of two virtual networks in RPM algorithm is unbalance. Compared with BAM, DBM algorithm achieved 8.6% latency reduction and 6.3% throughput improvement.

**Fig. 4.** Average packet latency in uniform random traffic

Similarly, in transpose and random permutation patterns, the saturation throughput of DBM is the largest and BAM is the following. The advantage of performance of DBM is obvious. Relative to RPM, DBM achieved 24.9% and 12.7% latency reduction and 71.4% and 75% throughput improvement respectively. Since the traffic patterns of transpose and random permutation cause the unbalancing load of different dimensions easier than uniform random, for RPM algorithm, the performance in these patterns is worse than uniform random.

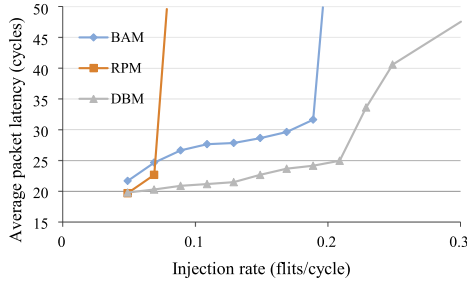


Fig. 5. Average packet latency in transpose traffic

The simulation in bit rotation pattern shows the similar result. The performance of RPM is the worst in three algorithms. It can be seen that in general, latency will increase as load (injection rate) increases. RPM is clearly not scalable as latency will dramatically increase with traffic load. Both DBM and BAM perform better than RPM. So, we only compare DBM with BAM in the following simulation.

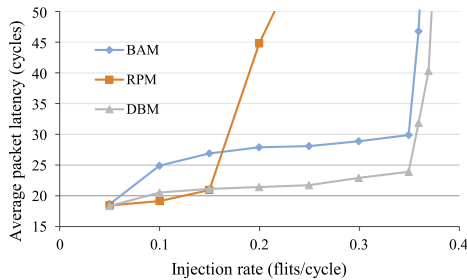


Fig. 6. Average packet latency in random permutation traffic

4.3 Impact of Buffer Size

The bit rotation traffic is considered to be the worst case for all the networks under study. We study network performance with different buffer under this traffic pattern.

The performance with different buffer sizes are showed in Fig. 8. DBM and BAM’s algorithm both improve given larger buffer sizes. Comparatively, DBM performance is more advantage. For example, relative to DBM, when the buffer size is 4, 6 and 8 respectively, DBM performance achieved 8.6%, 8.1% and 14.2% latency reduction and 6.3%, 8.8% and 12.2% throughput improvement.

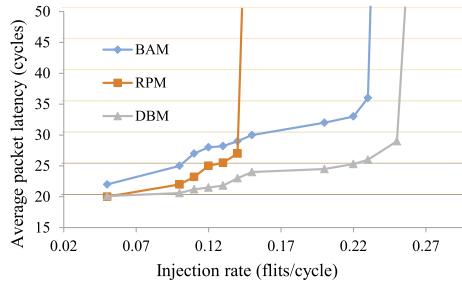


Fig. 7. Average packet latency in bit rotation traffic

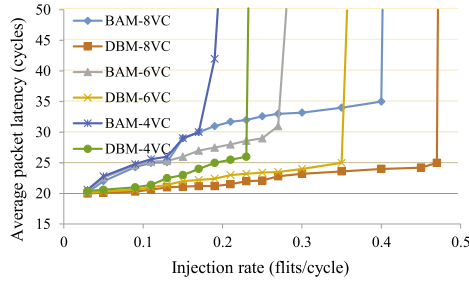


Fig. 8. Average packet latency with different buffer size

4.4 Scalability of Network Size

In 2D mesh networks, DBM exhibits better scalability than BAM with respect to network size. The performance with different network sizes are showed in Fig. 9. Using the uniform random pattern, we compare latencies by increasing the number of nodes in each dimension from 16 to 64. When the number of nodes from 16 to 64, relative to BAM, DBM achieved latency reduction from 13.8% to 18.1% and throughput improvement from 8.6% to 9.8%.

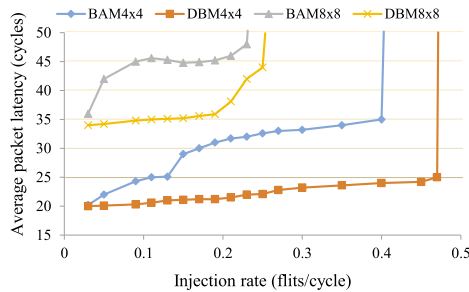


Fig. 9. Average packet latency with different network size

4.5 Discussion

DBM can improve the network performance by employing more restrictive flow control scheme than that in BAM’C the more remaining dimensional directions of replicated packet, the more buffer space in the next hop required. Although such bias may raise the chances of conflicts, it encourages packets to use more paths and enables DBM to unburden the network by balancing traffics.

In contrast, BAM in Duato’s framework, a packet is free to enter any queue as long as credit available. This freedom tends to form a locked ring in busy networks and requires the escape channels to break the deadlock which is very likely to end up with the “hot-spot” problem causing performance degradation.

More importantly, DBM promises the adaptively throughout the trip of a packet. However, in BAM, a packet has to enter into the DOR-based escape channel for deadlock avoidance when conflicts occur. As a consequence, within a typical injection rate range, traffic congestion tends to be alleviated in the networks using DBM. This explains why DBM is more adept at preventing the networks from performance degradation while the injection rate increases or traffic patterns become more adverse.

Understandable, a network’s overall performance is determined by the “worst case” routing. In fact, buffer size increment helps DBM attenuate the “worst case” impact, which happens when a queue becomes full. In other words, a larger buffer size makes a queue less likely to become full.

Specifically, the flow control DBMFC keeps the number of “bubbles” as balanced as possible in queues, and regardless of the buffer size, this mechanism remains functioning. However, this is not the case for BAM. The “worst case” in BAM is the “hot-spot” followed by DOR routing. Unfortunately, larger buffer size makes a queue capable of accommodating more packets implying that the queue contains more packets when it becomes full. Thus, more packets may enter into the DOR-based escape channel, which prolongs the average latency.

5 Conclusions and Future Work

In this paper, we design the DBM multicast routing strategy for 2D mesh networks. DBM algorithm provides a new way to implement high performance multicasting routing in interconnection networks. We prove that the proposed algorithm is deadlock-free while enabling minimal path and fully-adaptive multicast routing. Moreover, we complete some comparative work against RPM and BAM algorithms, which indicates that DBM can achieve more performance and scalability improvement under synthetic workloads. In the future work, we will study the micro-architecture of router based on DBM.

Acknowledgment. We thank the anonymous reviewers for their valuable feedback. We gratefully acknowledge members of Tianhe interconnect group at NUDT for many inspiring conversations. This project was partially supported by the National Key R&D Program of China under Grant No. 2018YFB0204300, and in part by NSFC. 61802416 and School Research Project of National University of Defense Technology under grants No. ZK20-18.

References

1. De Azevedo, M.M., Blough, D.M.: Fault-tolerant clock synchronization of large multicomputers via multistep interactive convergence. In: Proceedings of the 16th International Conference on Distributed Computing Systems, pp. 249–258 (1996)
2. Duato, J., Pinkston, T.M.: A general theory for deadlock-free adaptive routing using a mixed set of resources. *IEEE Trans. Parallel Distrib. Syst.* **12**(12), 1219–1235 (2001)
3. Furber, S., Galluppi, F., Temple, S., Plana, L.A.: The spinnaker project. *Proc. IEEE* **102**(5), 652–665 (2014)
4. Jerger, N.E., Peh, L., Lipasti, M.H.: Virtual circuit tree multicasting: a case for on-chip hardware multicast support. In: Proceedings of the International Symposium on Computer Architecture, pp. 229–240 (2008)
5. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 86–96 (2013)
6. Li, K., Schaefer, R.: A hypercube shared virtual memory system. In: Proceedings of the International Conference on Parallel Processing, pp. 125–132 (1989)
7. Ma, S., Jerger, N.E., Wang, Z.: Supporting efficient collective communication in NoCs. In: Proceedings of the IEEE International Symposium on High-Performance Computer Architecture, pp. 1–12 (2012)
8. Mckinley, P.K., Xu, H., Kalns, E.T., Ni, L.M.: Compass: efficient communication services for scalable architectures. In: Proceedings of the ACM/IEEE Conference on Supercomputing, pp. 478–487 (1992)
9. Navaridas, J., Lujan, M., Miguelalonso, J., Plana, L.A., Furber, S.: Understanding the interconnection network of spinnaker. In: Proceedings of the 23rd International Conference on Supercomputing, pp. 286–295 (2009)
10. Rodrigo, S., Flich, J., Duato, J., Hummel, M.D.: Efficient unicast and multicast support for CMPS. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, pp. 364–375 (2008)
11. Wang, L., Jin, Y., Kim, H., Kim, E.J.: Recursive partitioning multicast: a bandwidth-efficient routing for networks-on-chip. In: Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip, pp. 64–73 (2009)
12. Xiao, C., Yang, Y., Zhu, J.: A sufficient condition for deadlock-free adaptive routing in mesh networks. *IEEE Comput. Archit. Lett.* **14**(2), 111–114 (2015)
13. Xu, H., Mckinley, P.K., Ni, L.M.: Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputer's. *J. Parallel Distrib. Comput.* **16**(2), 172–184 (1992)



MPLEG: A Multi-mode Physical Layer Error Generator for Link Layer Fault Tolerance Test

Xingyun Qi^(✉), Pingjing Lu, Jijun Cao, Yi Dai, Mingche Lai, and Junsheng Chang

College of Computer, National University of Defense Technology, Changsha, Hunan, China
qi_xingyun@mail.nudt.edu.cn

Abstract. In the design of high-speed communication network chips, the fault-tolerant design of the link layer is among the most important parts. In the design process, the link layer fault tolerance function need to be fully tested and verified. But it is far from enough to rely only on traditional case-by-case simulation. In order to test and verify this function completely, this paper proposes a configurable multi-mode physical layer error generation method implemented on chip: MPLEG (a Multi-mode Physical Layer Error Generator). With MPLEG, a desired bit error pattern can be generated at the physical layer in all stages of chip design, including simulation verification, FPGA prototype system verification, sample chip testing, and actual system running. The statistical analysis of the experimental results shows that MPLEG can generate an error pattern almost identical to the real link error. Meanwhile, MPLEG can perform relatively complete and efficient testing and verification of various functions of link layer fault tolerance.

Keywords: Physical link error · Error pattern analyze · Error generator on chip · Link layer fault tolerance test

1 Introduction

In high-speed digital communications, the traditional parallel transmission method has been unable to meet the ever-increasing communication bandwidth requirements, and high-speed serial transmission methods have emerged. In the serial communication mode, digital communication between two nodes is performed through high-speed serial differential signals. The transmitter encodes the parallel data and clock signal, and then converts it into a serial data stream through a SERializer/DESerializer (SERDES), and then sends it to the physical medium; the SERDES in the receiving end receives the serial data, then the serial data is converted into parallel data, and the clock of the receiving end is recovered by the clock and data recovery (CDR) circuit. According to the network layer structure, serial-to-parallel conversion, clock recovery, data encoding and decoding, and data transmission processes in serial digital communications all belong to the physical layer function [1]. In this process, the high-speed serial signal may cause errors in the transmission process due to various reasons. Under normal circumstances, there may be two reasons for the occurrence of bit errors: (1) due to the timing jitter, the actual signal transition is earlier or later than the ideal position; (2) due to amplitude noise,

the high level of the signal is lower than the reference threshold or the low level of the signal is higher than the reference threshold. Therefore, in a high-speed communication system, after the link layer receives data from the physical layer, one of the important functions is the fault tolerance processing, including data verification, error detection, error correction, and retransmission. The purpose of fault tolerance is to provide reliable data transmission services for the upper layers of the network.

In the design of communication systems, especially the design of high-speed network communication chips, in order to test the correctness of the link layer error processing system, it takes a lot of resources to verify and evaluate each function of the fault tolerance system. In the process, the most critical thing is to artificially generate various errors close to the real situation at the physical layer, so as to cover most error testing scenarios. The physical layer errors generated for the purpose of performing link layer fault-tolerant tests play a vital role in the correctness and completeness of the final system verification.

There are currently two commonly used link fault tolerance test methods: simulation test and real system test. The simulation test method is using VCS [2]/Xcelium [3] and other front-end simulation tools to simulate the RTL design, and manually inserting the errors expected by the designer on the link. But the test data of this method is limited, and the subjective bias of the designer may not be able to reproduce the error scenario under the real environment, and its test coverage is limited. The real system test method is to build a real hardware test environment, apply interference on the physical link, and then test the fault tolerance design of the system. The real system test method can indeed reproduce the real error scenario, but the cost is too high, and it is not easy to find the error. At the same time, the real system test method generally needs to be carried out after the chip has been taped out, so it is difficult to perform a complete test at the design stage. Therefore, to design a link fault tolerance test method which is capable of simulating real physical link errors inside the chip is necessary, so as to carry out various boundary and abnormal tests on link transmission in the design stage of high-speed communication chips.

Aiming at the above problems, this paper first collects the bit errors on the real link, analyzes its distribution characteristics, and designs an on-chip error generation method of physical layer - MPLEG according to the characteristics. Using MPLEG, a physical layer error similar to the real situation can be generated inside the high-speed communication chip, which is used to test the fault-tolerant design inside the chip.

The rest of this paper is organized as follows: Sect. 2 introduces the relevant research background; Sect. 3 collects and analyzes the bit error characteristics on the actual physical link; Sect. 4 presents the method of MPLEG; Sect. 5 evaluates the effect and performance of on-chip error generated by MPLEG; Sect. 6 summarizes the work and draws relevant conclusions.

2 Related Works

Most researches about physical link error pattern are focused on wireless communications [10–14] and free space optical links [15]. In [10], the target packet error sequences are generated by a simulator with a typical urban (TU) channel and co-channel interference. A general design procedure of a generative model is then proposed by using

a properly parameterized and sampled deterministic process with a threshold detector and two parallel mappers. A generative error model that can generate packet-level error sequences with predicted burst error statistics is proposed in [11], which can generate errors similar to those of error sequences obtained from real wireless systems. [12, 13] both proposed a link error prediction method respectively. Both methods are shown to have accuracy within a few tenths of a dB under a wide range of modulation schemes, coding rates and channel types. These methods are then extended to handle more advanced link enhancements such as hybrid ARQ and Alamouti encoding [14]. Wu et al. [16] studied chip error patterns in IEEE 802.15.4, and found out that there are four major error patterns. They proposed a simple yet effective method based on the chip error patterns to infer the link condition.

For high speed serial communication, there are currently two methods for evaluating the signal quality: simulation analysis and worst-case link analysis. Among them, the simulation analysis method generally uses tools such as SPICE for pseudo-random data simulation, which usually uses a Pseudo-Random Bit Sequence (PRBS) generator to simulate the link channel. The worst-case link analysis method calculates the worst eye diagram of the system by analyzing the channel response and noise model, and estimates the worst bit error rate of the system [4–6]. Both of these two methods have advantages and disadvantages. The simulation analysis method needs to generate a large amount of random test cases to cover various transmission scenarios of the link, therefore it is difficult to ensure complete coverage of various test scenarios [8]. The worst-case analysis method only focuses on the link in the worst case. Therefore, the result is too pessimistic, which will put more strict requirements on the design [7]. Some researchers [8] studied the error conditions of high-speed links from the perspective of probability statistics, and proposed a new bit error rate analysis method to analyze the eye diagram of any pattern under the influence of link ISI and crosstalk. Then the paper calculated the probability of each situation, and obtained the bit error rate eye diagram of the link according to the probability distribution at the receiving end. Dongwoo Hong et al. believes that the error in high-speed communication mainly comes from data jitter, including random jitter (RJ) and deterministic jitter (DJ) [9]. This paper proposes a method to theoretically analyze the RJ and DJ of signals in the data at the receiving end, and gets the bit error rate on high-speed serial links.

Some researchers assumes that the ideal noise sources follow the Gaussian distribution. For real noise source generation, two uniformly distributed pseudo-random signals are generated, which are then transformed into actual noise sources through function transformation, and implemented on the FPGA chip. However, this method is only compared with the theoretical Gaussian distributed noise source, but not with the actual noise on the real link.

All the researches above have only analyzed and estimated the noise on the physical link. Some have only designed an ideal noise source, but they have not been compared with the noise on the actual link. Moreover, most of the current researches focus on the theoretical analysis of the noise and interference on the serial link. These researches lack statistics and analysis of interface data error between physical layer and link layer, which is precisely the link layer fault tolerance design should pay attention to. Some researches

This paper is geared towards the fault-tolerant design of the link layer. It collects and analyzes the true bit errors of the data on the interface between the physical layer and the link layer. Based on the analysis, a physical layer bit error generator inside the chip is proposed, which can generate various modes of bit error close to the actual bit error mode. It can be used to fully test the fault-tolerant design of the link layer.

3 Statistics of Real Bit Error in Physical Links

In order to analyze the real bit error characteristics on the physical link in the high-speed serial communication, we use an FPGA chip evaluation board with a high-speed serial communication interface to design a hardware system that can collect and analyze the physical layer data transmission bit errors. The system communicates between two FPGA chips through a high-speed serial interface. The transmitter continuously sends out pseudo-random sequences as test data, while the receiver continuously receives data and checks the correctness of the data. The system structure is shown in Fig. 1.

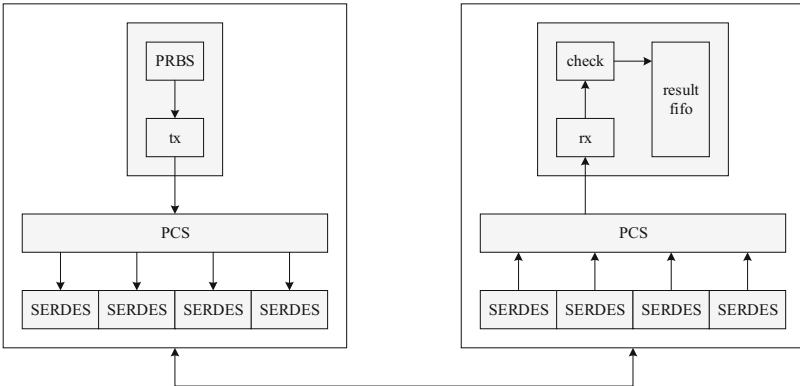


Fig. 1. The structure of the physical layer error capture

At the receiving end, each received data is checked for correctness. If data error occurs during transmission, the receiver will record the specific error information, including the expected data, the wrong data, the time interval between this error and the previous error, and the total number of error bits. Based on these information, the distribution characteristics of error can be figured out, and the error model can be obtained.

In the experiment, we chose a fiber with errors with which the two FPGA are connected in the test system. The FPGA chips chosen in the experiment are Xilinx Virtex 7 2000T devices. In the evaluation board, each port uses a 4lane GTX, with the rate of 1.25 Gbps per lane. The test system based on FPGA evaluation boards is illustrated in Fig. 2.

The test continues about 30 min, and the data is continuously sent at the maximum rate. The receiver detects the correctness of the data and records the error information. The test results are shown in Table 1.



Fig. 2. Link error test system based on FPGA

Table 1. Statistics of physical link real error information

Items	Value
Total amount of the test data	7.5×10^{12} bits
Total error bits	134128 bits
Average error rate	1.788×10^{-8}
Average interval of 2 error (take 256bit data as the unit)	4.4706×10^5

We analyzed the time interval between two adjacent errors, and take statistics of the interval distribution. Figure 3(a) is a distribution diagram of different error intervals. It can be seen that the number of errors with an interval of 1 (that is, two adjacent errors) is significantly larger than the errors of other intervals. The number of errors of other intervals gradually decreases with the increasing of interval. We can see that adjacent error with an interval of 1 is the burst error. Figure 3(b) is the error distribution diagram with the error interval greater than or equal to 2, and Fig. 3(c) is the error distribution diagram with the error interval of 1. It can be seen from the figure that the bit errors in the actual link are actually the superposition of burst errors and uniform errors. That is, Fig. 3(a) is the superposition of Fig. 3(b) and Fig. 3(c).

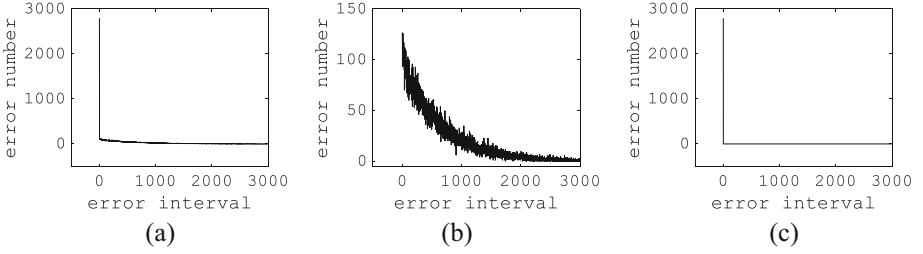


Fig. 3. Number of errors in different intervals

4 Multi-mode Physical Layer Error Generator - MPLEG

By analyzing the bit error characteristics of the real physical link in above experiment, we find that the bit error on the actual link is a mix form of uniform bit error and burst bit error. According to this feature, we design a configurable multi-mode physical layer error generator - MPLEG. The error generator is located between the PCS (Physical Coding Sublayer) and SERDES at receiver, and is used to inject a specific pattern of error sequences into the data from SERDES. According to this structure, an error generating module (error_insert) is designed at the receiving end of each lane to insert a specific error bit sequence into the parallel data given by SERDES, as shown in Fig. 4.

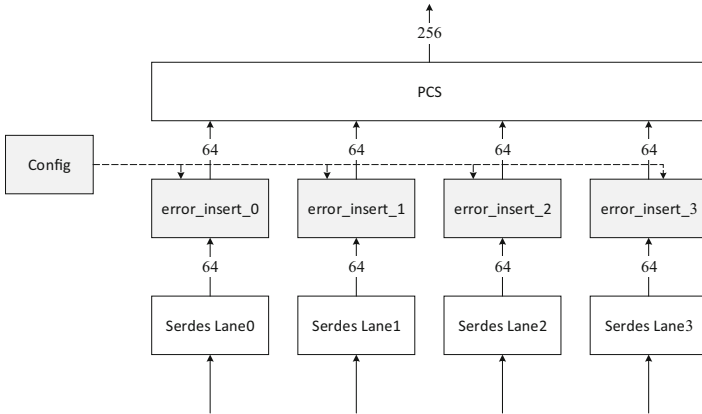


Fig. 4. Structure of error generator

In Fig. 4, there are 4 lanes in each network port. After serial-to-parallel conversion, each SERDES provides 64-bit-wide parallel data bits to PCS. The 4 lanes are bound together to provide a 256-bit-wide data path. Under the control of the Config module, each error_insert module reverses every bit in the incoming 64-bit data according to certain error generation rules. For example, if the link error rate is set to 10^{-8} , then the probability of error per bit is $1/10^8$. That is, in that lane, a bit flips once about every 108 clock cycles on average (0 to 1 or 1 to 0). Each error_insert contains 64 error-generating

units (named `err_element`), and each error-generating unit generates errors in a certain bit of 64-bit data. Therefore, all data bit errors are independent of each other, as shown in Fig. 5.

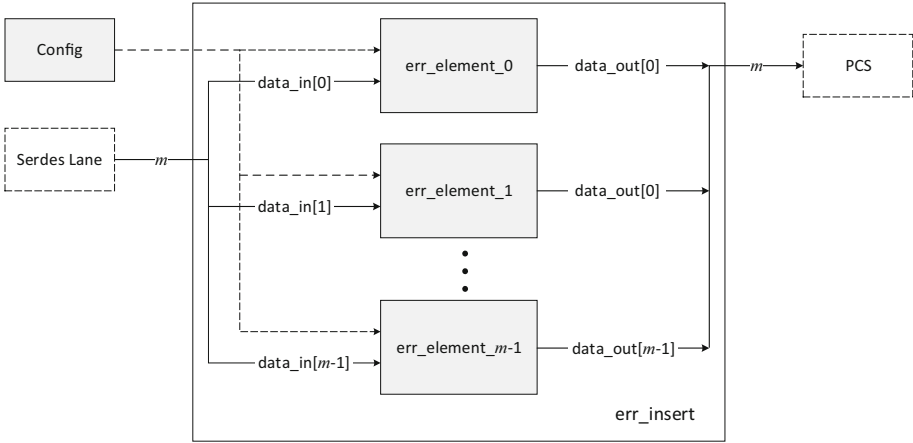


Fig. 5. Structure of `err_insert` module

The function of the error-generating unit is to generate all kinds of error patterns according to the configuration. Based on the previous analysis of the error characteristics on the physical link, the error-generating unit should provide multiple configurations to achieve a variety of error distribution characteristics such as uniform error, burst error and mixed error. From the perspective of functional components, the error-generating unit is divided into a uniform error generator, a burst error generator, and a synthesizer, which are used to generate uniform errors, burst errors, and multiple error modes. The uniform error generator generates uniform errors according to the input signal `cfg_err_rate`. The error rate of the uniform error is `cfg_err_rate/232`. The burst error generator generates two kinds of error with different error rates (high error rate and low error rate) according to the input signals `cfg_err_burst_low_period`, `cfg_err_burst_high_period`, `cfg_err_rate_low` and `cfg_err_rate_high`. The high error rate is `cfg_err_rate_high/232` and the low error rate is `cfg_err_rate_low/232`, and the durations of high error rate and low error rate are `cfg_err_burst_high_period` and `cfg_err_burst_low_period`, respectively. In this way, when different configuration are given, different burst errors can be generated. According to the configuration, the synthesizer combines the previously generated uniform errors and burst errors into a mixed error mode, and the physical layer error is the result of a mixture pattern of uniform errors and burst errors. The structure of error-generating unit is shown in Fig. 6.

The processing flow chart of error-generating unit is shown in Fig. 7.

When the configuration signal `cfg_err_mode` is 0, no error is generated, that is, output data `data_out` is equal to input data `data_in`. When `cfg_err_mode` is 1, errors are generated in uniform mode, that is, the output data `data_out` is the inverse of the input data `data_in` with certain probability. When `cfg_err_mode` is 2, the error-generating units

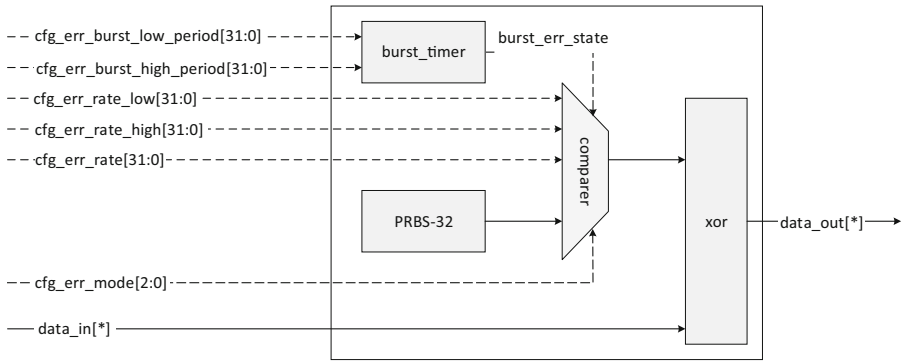


Fig. 6. Structure of error-generating unit

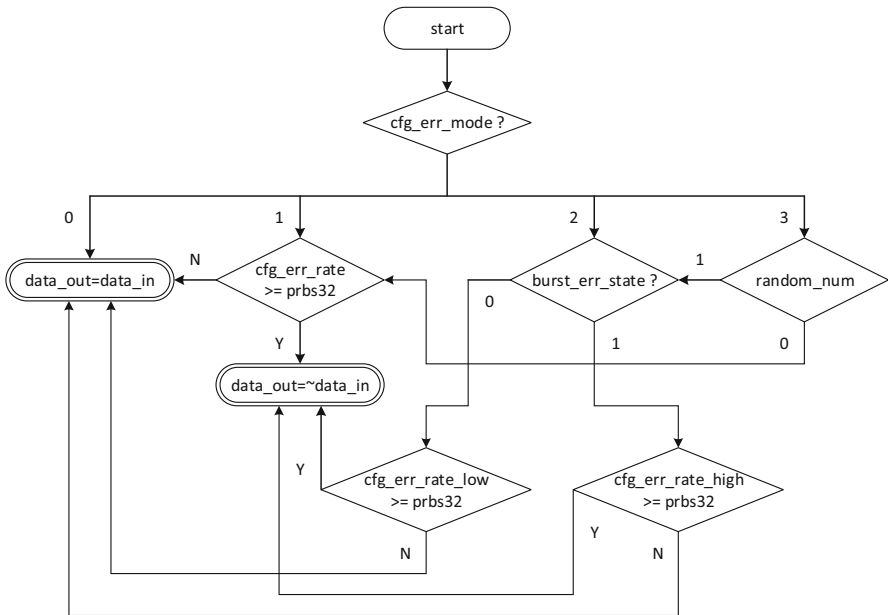


Fig. 7. The processing flow of error-generating unit

are in burst error mode, that means, two different periods are generated according to the configuration signal `burst_time`, which are corresponding to the high error rate time and low error rate time. In the high error rate period, the error rate is `cfg_err_rate_high`, and in the low error rate period, the error rate is `cfg_err_rate_low`. That is, there exists bit error in output data `data_out` compared with `data_in`. The error rate changes between two different preset bit error rates according to the configuration. In this way, the burst error mode can be achieved. When `cfg_err_mode` is 3, the error-generating units are in mix mode. In this mode, a 1-bit random number (`random_num`) is generated to determine whether it is currently in uniform error mode or burst error mode randomly. When the

random number is 0, it is in a uniform error mode, and when the random number is 1, it is in a burst error mode.

In this way, an error generating module (error_insert) contains 64 independent error-generating units, and each error-generating unit generates an error to one of the data bit. Figure 8 shows the structure of an error_insert module.

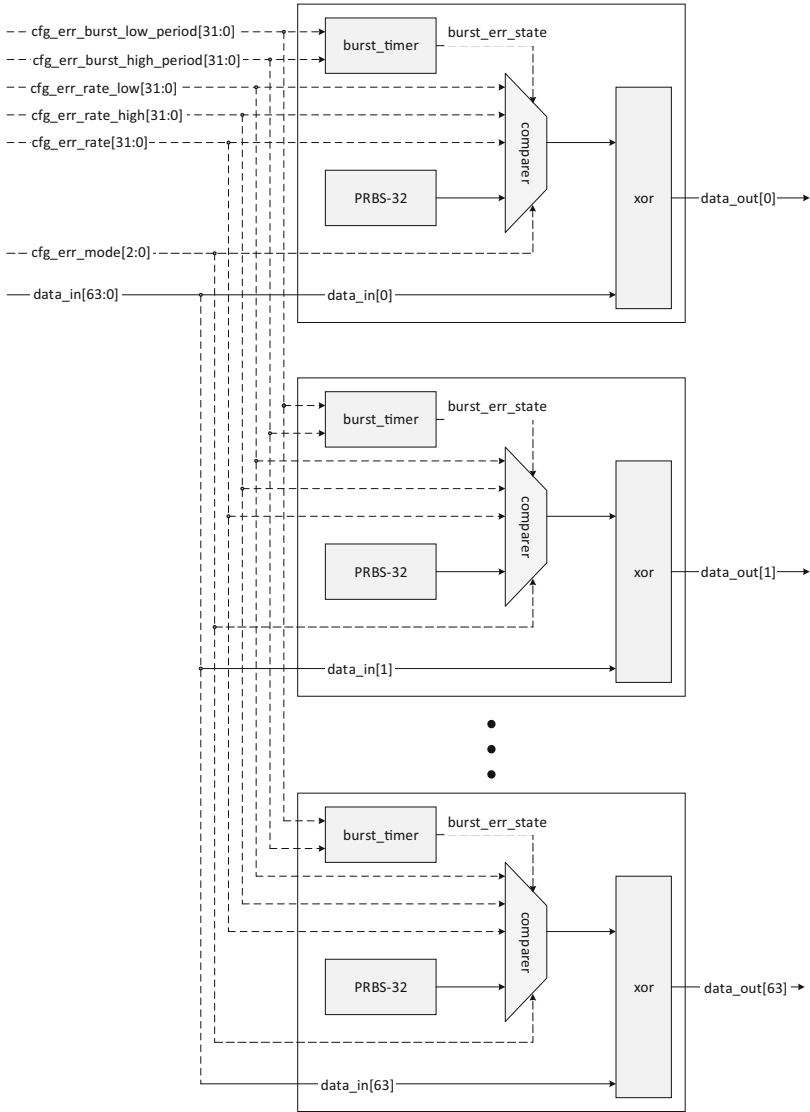


Fig. 8. Structure of an error_insert module

5 Evaluation

We will analyze and evaluate the errors at the physical layer generated by the physical layer error generator proposed above.

5.1 Evaluation Criterion

The error data recorded in the experiment is an n -tuple, where n is the number of records. In the experiment, we take $n = 65535$. Therefore, the actual error data is recorded as

$$R = [r_1, r_2, \dots, r_{65535}]$$

The error data generated by the error generator described in this paper is recorded as

$$G = [g_1, g_2, \dots, g_{65535}]$$

Where $r_i, g_i = 0, 1, 2, \dots$, means the interval between the current error data and the last error data.

For example, if a certain error data is recorded as [74, 382, 0, 1673, 258, 4, 67], it means that a total of seven errors occurred during the recording time. There are 74 correct data between the first error data and the first data, and 382 correct data between the second error data and the first error data, and 0 correct data between the third error data and the second error data (which means the second and the third error data are continuous), ..., and 67 correct data between the seventh error data and the sixth error data.

In order to evaluate the difference between R and G , we define the distribution distance between two sets of data records below.

Let $s = 1, 2, 3, \dots$, and s is the dividing scale. Divide the record R and G according to the following algorithm, and calculate their distribution distance:

- (1) Let $m = \max\{R, G\}$
- (2) Initialize 2 $\lfloor \frac{m}{s} \rfloor$ -dimensional vectors

$$V_r \left[0 : \left\lfloor \frac{m}{s} \right\rfloor - 1 \right] = 0$$

$$V_g \left[0 : \left\lfloor \frac{m}{s} \right\rfloor - 1 \right] = 0$$

- (3) for $i = 1, 2, \dots, 65535$

$$V_r \left[\left\lfloor \frac{r_i}{s} \right\rfloor \right] = V_r \left[\left\lfloor \frac{r_i}{s} \right\rfloor \right] + 1$$

$$V_g \left[\left\lfloor \frac{g_i}{s} \right\rfloor \right] = V_g \left[\left\lfloor \frac{g_i}{s} \right\rfloor \right] + 1$$

- (4) When the division scale is s , the average distribution distance between R and G is defined as

$$D_s(R, G) = \left| \frac{1}{\lfloor \frac{m}{s} \rfloor} \sum_{i=0}^{\lfloor \frac{m}{s} \rfloor - 1} (V_r[i] - V_g[i]) \right|$$

and the accumulated distribution distance between R and G is defined as

$$E_s(R, G) = \frac{1}{\lfloor \frac{m}{s} \rfloor} \sum_{i=0}^{\lfloor \frac{m}{s} \rfloor - 1} |V_r[i] - V_g[i]|$$

When $s = 1$, V_r and V_g indicate the number of error messages with different error intervals. That is, $V_r[0]$ represents the number of 0s in R , i.e. the number of error packets with an error interval of 0. $V_r[1]$ represents the number of 1s in R , i.e. the number of error packets with an error interval of 1.

When $s > 1$, it is equivalent to dividing the data in R and G into $\lfloor \frac{m}{s} \rfloor$ groups respectively with the interval s from small to large. Then the number of data in each group are counted. Thus, $V_r[i]$ indicates the number of data whose value is $i \cdot s \sim (i + 1) \cdot s - 1$ in R . For example, $V_r[0]$ represents the number of data in R with a value of $0 \sim s - 1$, that is, the number of error packets with an error interval of $0 \sim s - 1$. $V_r[1]$ represents the number of data in R with a value of $s \sim 2s - 1$, that is, the number of error packets with an error interval of $s \sim 2s - 1$.

When s is constant, $D_s(R, G)$ and $E_s(R, G)$ is the distance between vectors R and G . The smaller $D_s(R, G)$ and $E_s(R, G)$ is, the smaller the difference between R and G is.

5.2 Evaluation of Generated Error Data

We use the FPGA test system in Sect. 3 to evaluate the error data generated by the physical layer error generator. Firstly, the FPGA system is tested with an optical fiber that has bit errors, and a set of original error record R with the elements number of 65535 is obtained. Then the record R is analyzed to calculate the bit error rate. Based on this error rate, appropriate parameters of the error generator are set. Then the FPGA system is tested again using a normal fiber. The error results of the test are recorded as G . According to the analysis method mentioned before, the difference between R and G is evaluated for the two sets of error records.

According to the algorithm presented in 5.1, we take different division scales $s = 1, 2, 3, \dots, 100$ to analyze R and G respectively. Figure 9 shows the curves of V_r and V_g when s is 1, 3, 5, 10, 20, 30, 50, 80 and 100. That is, the distribution of R and G under different division scales s . It can be seen from the figure that the distribution of the error interval of R and G are basically coincident. It indicates that the result of MPLEG are close to the result of actual physical link errors in different scale s , and the physical layer errors generated by MPLEG can basically fit the error pattern under actual conditions.

We also calculated the distance between actual error record R and generated error record G at different division scales ($s = 1, 2, 3, \dots, 5000$). Figure 10(a) and (b) show the curves of $D_s(R, G)$ and $E_s(R, G)$ with the increasing of s , respectively. It can be seen

from the figure that no matter what the value of s , the curve of $D_s(R, G)$ is basically 0. That is, the average distance between R and G remains very low, which indicates that the distribution curves of R and G have significant similarities. At the time, $E_s(R, G)$ increases with the increase of s . When $s > 1000$, $E_s(R, G)$ gradually decreases with the increase of s . Under different values of s , the average value of $E_s(R, G)$ is 684.5, which is still at a low level, indicating that the difference between R and G is small.

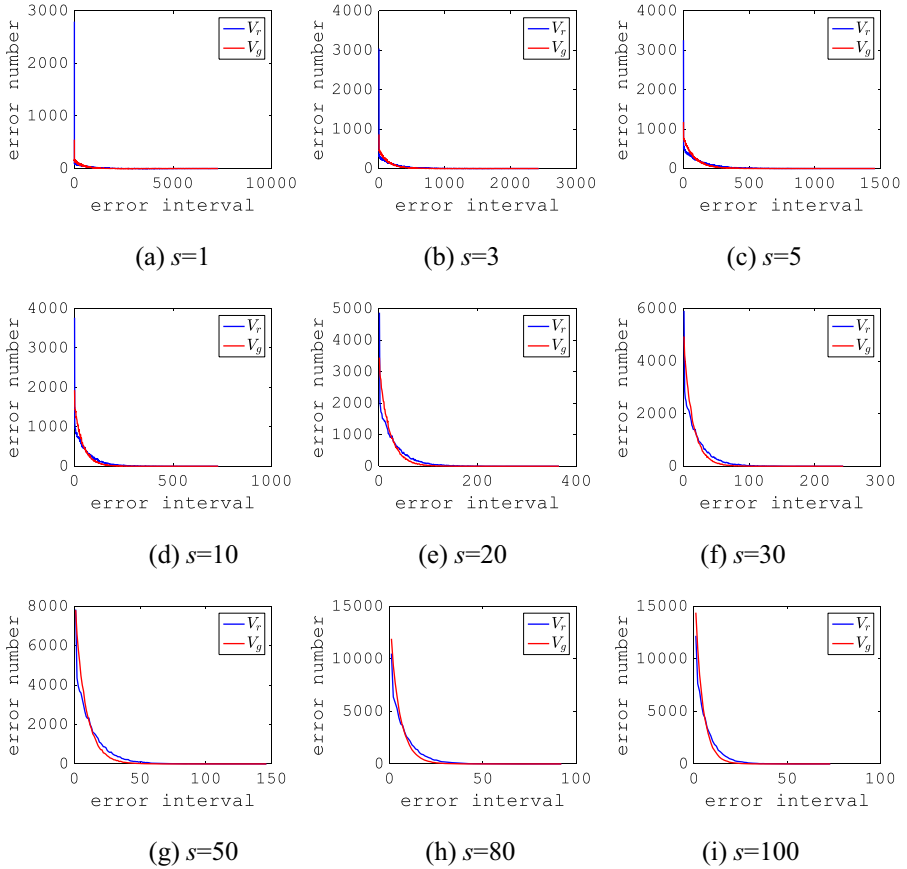


Fig. 9. V_r and V_g in different s

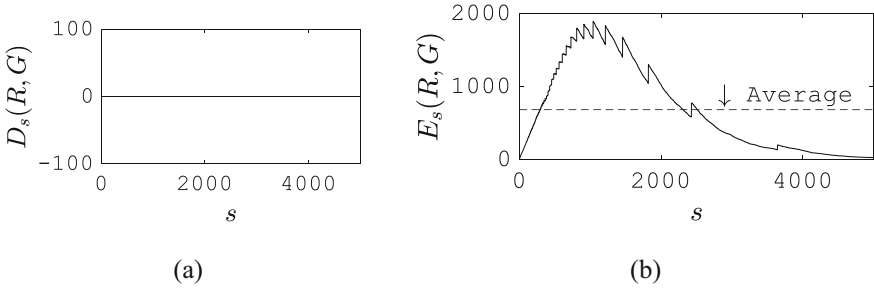


Fig. 10. Distance between R and G

5.3 Actual Link Layer Fault Tolerance Test

The purpose of MPLEG in this paper is to facilitate the test of the fault tolerance in the link layer. In order to verify whether the function of MPLEG proposed in this paper could test the fault tolerance function of the link layer comprehensively and reasonably, we use fiber with poor link quality and normal fiber with the error generator to test a same network environment respectively. Then we compare the performance differences of link layer fault tolerance in these two different environments. Table 2 shows the number of link layer retransmissions per second in the two error scenarios under different bit error rates.

Table 2. Link layer retransmission number in different error rate

Error mode	Number of error rate			
	$3 * 10^{-9}$	$2 * 10^{-8}$	$1 * 10^{-7}$	$3 * 10^{-6}$
Actual physical layer error	147	282	1218	14038
MPLEG	164	298	1176	14357

It can be seen from the table that MPLEG has an impact on the fault tolerance processing of the link layer, which is basically the same as the impact of the physical layer errors under the real link on the link layer. Under the two methods, the numbers of retransmissions in the link layer are basically equal within a certain period of time. So the transmission qualities of the two situation are same from the view of link layer. We can simulate the bit error scenario on the real physical link and test the link layer fault tolerance function using the physical layer error generation method in this paper.

6 Conclusion

This paper aims at the difficulty of effectively making errors on the physical link during the design and verification process of the current high-speed interconnection network chip, which makes it impossible to fully test and verify the fault tolerance function of the

link layer. According to the actual error characteristics of the link, a configurable multi-mode physical layer error generation method - MPLEG is proposed. Using MPLEG, a configurable error mode that combines burst errors and uniform errors can be generated based on actual link error patterns and scenarios. By comparing the actual physical link error with the generated error, we find that MPLEG is similar to actual link layer error in many aspects such as error mode, error distribution characteristics and impact on the actual link layer fault tolerance processing. MPLEG can simulate the errors on the physical link in the chip, and provide strong support for the test of the link layer fault tolerance function.

References

1. Tanenbaum, A.S, Wetherall, D.J.: Computer Networks. 5 edn. (2010)
2. Synopsys VCS. <https://www.synopsys.com/verification/simulation/vcs.html>. Accessed 21 Apr 2020
3. Cadence Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html. Accessed 21 Apr 2020
4. Casper, B.K., Haycok, M., Mooney, R.: An accurate and efficient analysis method for multi-Gb/s chip-to-chip signaling schemes. In: IEEE Symposium on VLSI Circuits Digest of Technical Papers, Honolulu, HI, USA (2002)
5. Casper, B., O'Mahony, F.: Clocking analysis, implementation and measurement techniques for high-speed data links—a tutorial. *IEEE Trans. Circ. Syst.* **56**(1), 17–39 (2009)
6. Ren, J., Oh, D.: Multiple edge responses for fast and accurate system simulations. *IEEE Trans. Adv. Packag.* **31**(4), 741–748 (2008)
7. Chada, A.R., Wu, S., Fan, J., et al.: Efficient complex broadside coupled trace modeling and estimation of crosstalk impact using statistical BER analysis for high volume. high performance printed circuit board designs. In: IEEE 63rd Electronic Components and Technology Conference (ECTC), Las Vegas, NV, pp. 2095–2101 (2013)
8. Li, Y.: BER performance analysis of high-speed parallel link. Master's Thesis of XiDian University (2015)
9. Hong, D., Ong, C.-K., Cheng, K.-T.: Bit-error-rate estimation for high-speed serial links. *IEEE Trans. Circ. Syst.—I: Regul. Pap.* **53**(12), 2616–2627 (2006)
10. Wang, C.-X., Xu, W.: Packet-level error models for digital wireless channels. In: Proceeding of IEEE ICC 2005, Seoul, Korea, pp. 2184–2189, May 2005
11. Salih, O.S., Wang, C.-X., Mesleh, R., Ge, X., Yua, D.: Predicting burst error statistics of digital wireless systems with HARQ. In: 9th International Wireless Communications and Mobile Computing Conference (IWCMC) (2013)
12. Lampe, M., Rohling, H.: PER-prediction for PHY mode selection in OFDM communication systems. In: Proceeding of IEEE GLOBECOM 2003, vol. 1, pp. 25–29, December 2003
13. Ericsson: System-level evaluation of OFDM - further considerations. 3GPP TSG-RAN WG1#35, R1-031303, pp. 17–21, November 2003
14. Blankenship, Y.W., Sartori, P.J., Classon, B.K., Desai, V., Baum, K.L.: Link error prediction methods for multicarrier systems. In: Proceeding of VTC 2004-Fall, Los Angeles, USA, pp. 4175–4179, September 2004
15. Ansari, I.S., Yilmaz, F., Alouini, M.-S.: Performance analysis of free-space optical links over Málaga (M) turbulence channels with pointing errors. *IEEE Trans. Wirel. Commun.* **15**(1), 91–102 (2016)
16. Wu, K., Tan, H., Ngan, H., Liu, Y., Ni, L.M.: Chip error pattern analysis in IEEE 802.15.4. *IEEE Trans. Mob. Comput.* **11**(4), 543–552 (2012)

Accelerator-Based, Application-Specific and Reconfigurable Architecture



GNN-PIM: A Processing-in-Memory Architecture for Graph Neural Networks

Zhao Wang¹, Yijin Guan², Guangyu Sun^{1(✉)}, Dimin Niu², Yuhao Wang²,
Hongzhong Zheng², and Yinhe Han³

¹ CECA, Peking University, Beijing, China
gsun@pku.edu.cn

² Alibaba DAMO Academy, Hangzhou, China

³ Institute of Computing Technology,
Chinese Academy of Sciences, Beijing, China

Abstract. Graph neural networks (GNNs) have attracted increasing interests in recent years. Due to the poor data locality and huge data movement during GNN inference, it is challenging to employ GNN to process large-scale graphs. Fortunately, processing-in-memory (PIM) architecture has been widely investigated as a promising approach to address the “Memory Wall”. In this work, we propose a PIM architecture to accelerate GNN inference. We develop an optimized dataflow to leverage the inherent parallelism of GNNs. Targeting the dataflow, we further propose a hierarchical NoC to perform concurrent data transmission. Experimental results show that our design can outperform prior works significantly.

1 Introduction

As the volume of non-Euclidean data [32] keeps growing, graph neural networks (GNNs) have attracted great attention because of their capability to express complex relationships and inter-dependency between objects. Inspired by the success of convolutional neural networks (CNNs) in computer vision domain, spatial-based GNNs have advanced rapidly in recent years. Most GNNs are composed of several “convolutional” layers, as those in CNNs [11, 19].

The “convolution” operation in GNN can be roughly divided into two phases [30]. **Aggregation Phase** aggregates nodes’ information from their multi-hop neighbours by pointer-chasing operations. This phase incurs intensive random memory accesses. **Handling Phase** feeds the aggregated features into a neural network to generate new features. Both computation and aggregation are regular in this phase. Having totally different processing patterns, the two phases consume considerable yet distinct resources. Thus, each of them may become the bottleneck of the whole system. We can tell that GNN processing encounters similar challenges as those in the graph processing and CNN inference, which have been well studied separately.

Recently, researchers propose to accelerate GNN processing by optimizing both software framework [20] and hardware architecture [30]. Ma et al. [20] developed

a programming framework for GNN processing by extending conventional edge-centric processing framework. Yan et al. [30] employs a hybrid architecture to handle the two phases separately. However, the size of graph datasets for GNNs keeps increasing in many applications, such as, social networks and e-commerce transactions [11, 22, 28]. As a result, the requirements for large capacity and high bandwidth of the memory subsystem are further raised.

To address this challenge, the processing in memory (PIM) architecture has been considered as a promising solution. It can greatly alleviate the overheads of data movement by offloading computation tasks to the storage. PIM architectures have been extensively explored for both DNN models and graph processing applications [8, 12, 24, 27]. However, it is inefficient to directly employ prior works for GNNs. For example, PRIME [8] proposes a comprehensive design of processing elements for NN processing on ReRAM. But its processing dataflow is not suitable for GNNs, and the bus-based communication among PEs may cause significant performance loss during the random sampling in the aggregation phase. Customized for graph processing, GraphR [27] addresses the random access problem in aggregation phase by employing edge-center processing model. However, the architecture for a conventional graph task lacks the support for efficient tensor processing.

Taking both aggregation phase and handling phase into consideration, we propose a PIM architecture equipped with a carefully designed NoC, called GNN-PIM. Our architecture employs SAGA [20] as the programming model and provides efficient PIM implementation. In GNN-PIM, we first leverage the computing capability of PIM architecture to support operations in handling phase. Then, we design a hierarchical interconnection network for efficient data movement in the aggregation phase. To the best of our knowledge, this is the first PIM accelerator proposed for GNNs. The contributions of this work are summarized as follows:

- By exploiting the inherent parallelism of GNNs, we propose a PIM architecture called **GNN-PIM** to accelerate the inference.
- We propose an optimized dataflow to map GNN inference efficiently on GNN-PIM, which is compatible with SAGA programming model.
- To facilitate the dataflow, we develop a hierarchical NoC providing high-bandwidth for data transmission.

The rest of this paper is organized as follows. In Sect. 2, we briefly review the computation of GNN inference with SAGA model. In addition, the basics of PIM architecture are introduced. In Sect. 4, we propose the GNN-PIM architecture and describe the execution dataflow on it. The hierarchical interconnection network of GNN-PIM is then presented in Sect. 3. Section 6 provides evaluation to demonstrate the efficiency of GNN-PIM, and Sect. 7 concludes this paper.

2 Background

In this section, we first introduce the operations of GNN inference and a dedicated programming model called SAGA. Then, we present basics about PIM designs.

2.1 GNN Inference and SAGA

GNNs are emerging neural networks that operate directly on graphs with non-Euclidean data. The idea of GNNs roots in CNNs and graph embedding [32]. Inheriting the ideas of parameter sharing from CNN and recursive execution from RNN, GNN convolution with kernel size one could be formulated as follows:

$$h_v^t = f(h_{v' \in N(v)}^{t-1}, h_v^{t-1}, I_e)$$

where $N(v)$ is the neighbourhood of v , h_v^t is the feature of vertex v in iteration t , and I_e are the labels of edges that are connected with v . To process the operations involved in GNN inference, vertices need to fetch features from their neighboring vertices. Since the neighbours of nodes are relevant to the topology of graph, random memory access occurs frequently. Such random memory access pattern can cause significant bandwidth waste and performance degradation.

Algorithm 1. GGCN in SAGA

input: $p = [W_H^l, W_C^l, W_e^l]$, $vertex^l$
output: $vertex^{l+1}$

- 1: $edge^l = \text{Scatter}(vertex^l)$
- 2: $acc = \text{ApplyEdge}(edge^l, p)$
- 3: **function** $\text{APPLYEDGE}(edge^l, p)$
- 4: $\eta = \text{sigmoid}(p.W_H^l \otimes edge^l.src + p.W_C^l \otimes edge^l.dst)$
- 5: **return** $\eta \odot edge^l.src$
- 6: **end function**
- 7: $accum = \text{Gather}(acc)$
- 8: $vertex^{l+1} = \text{ApplyVertex}(vertex^l, accum, p)$
- 9: **function** $\text{APPLYVERTEX}(vertex^l, accum, p)$
- 10: **return** $\text{ReLU}(p.W^l \otimes accum)$;
- 11: **end function**

To regulate the processing of GNN inference, Neugraph [20] proposes a programming model called SAGA. SAGA converts a primitive program to the edge-centric dataflow and alleviates random memory access. In SAGA, a GNN is decomposed and converted into *ApplyEdge* and *ApplyVertex* to handle edges and vertices, respectively. Combined with *Scatter* and *Gather* for data transmission, the processing of GNN inference is illustrated in Algorithm 1.

ApplyEdge takes parameters of the network (p) and scattered features ($edge$) as input, and generates the partially accumulated features (acc) which is further reduced by *Gather*. The inputs of *ApplyVertex* consist of $accum$, vertex data tensor ($vertex$), and p . *ApplyVertex* outputs new vertex features through a neural network, usually an MLP. *Scatter* and *Gather* perform data broadcasting before *ApplyEdge* and data gathering before *ApplyVertex*, respectively.

2.2 PIM Basis

PIM architecture breaks the “Memory Wall” by moving computation to where the data are stored. Thus, it is friendly for memory-intensive applications such as graph processing and deep neural networks. Recently, various PIM architectures have been proposed. Basically, these approaches can be categorized according to the computation unit.

For the first type, the computation unit is still based on the traditional logic, which is integrated close to or inside the memory array [6, 7, 10, 25]. This is also known as near-data-processing (NDP). For the second type, the computing unit is just built using the memory cell. Prior works have demonstrated that both traditional SRAM and DRAM technologies [1, 17, 23] and emerging memory technologies, such as ReRAM [24, 29, 31], MRAM [3–5], and PCM [18], can be leveraged for PIM designs.

Recently, many works [2, 8, 12, 21, 24, 26, 27] have revealed the potential benefits of using PIM architectures for both DNN and graph computing applications. However, these accelerators either lack the ability to handle graph-like data transmission pattern during GNN inference, or are unable to process NN computation efficiently. As a result, we propose the GNN-PIM architecture to simultaneously address the two drawbacks of prior works.

3 GNN-PIM Architecture

In this section, we introduce GNN-PIM’s hierarchical architecture. The top hierarchy is **Node Cluster**, composed of a number of **Nodes**. Each node owns several memory chunks and a set of **processing elements** (PEs).

3.1 Node

Node is the unit for performing computation on the sub-graphs. They are fundamental modules to build the whole architecture. As shown in Fig. 1, the micro-architecture of nodes includes several memory chunks and a processing core.

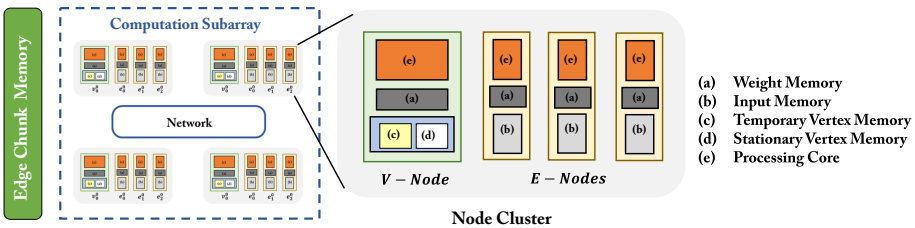


Fig. 1. Overall architecture of GNN-PIM

A node can be configured as either an E-Node or a V-Node, depending on the type of data stored in the memory chunk that they are working on. During GNN inference, E-Nodes are assigned with computation on edges, i.e. *ApplyEdge*. V-Nodes handle remaining tasks, consisting of *Scatter*, *Gather* and *ApplyVertex*. To simplify hardware design and provide more flexibility for mapping strategy, E-Node and V-Node share the same architecture and just differ in control logic which could be configured according to graph status.

We equip each V-Node with a processing core, weight memory for the storage of neural weights of *ApplyVertex*, along with temporary vertex memory and stationary vertex memory to store source vertex chunks and destination vertex chunks, respectively. An E-Node also consists of a processing core, a weight memory, as well as an input memory for storing input features of the layer that they are assigned to.

Matrix-Vector (MV) multiplication is the key operation of a processing core. It is used to process the most computing intensive functions in the phases of *ApplyEdge* and *ApplyVertex*. The computing architecture for MV operations have been extensively studied in prior works. Note that several non-MV functions are also needed, which are implemented with dedicated logic.

3.2 Node Cluster

Multiple Nodes can be grouped into a **Node Cluster** connected by NoC. The node cluster is composed of several nodes connected by a interconnection network. Some nodes are configured to be V-Nodes and the others are E-Nodes according to the sub-graph topology. In this way, more edges exist in the sub-graph, more nodes are configured to be E-Nodes. Since we employ a homogeneous design for the underlying hardware, all the nodes can be configured arbitrarily. As a result, GNN-PIM is able to configure nodes to balance the workloads according to the graph topology. Nodes in GNN-PIM work more like Multi-Processors System-on-Chip (MPSoC), transmitting data via the network-on-chip (NoC). In order to design an efficient NoC, we first need to understand the execution dataflow, which is introduced in the next section.

4 Execution Dataflow

We design an optimized dataflow that can map SAGA to GNN-PIM in a pipelining style. It is based on the phase sequence of *Scatter*, *ApplyEdge*, *Gather*, and *ApplyVertex*.

4.1 Mapping Strategy

As shown in Fig. 2, the whole graph is divided into multiple sub-graphs by dividing vertices into multiple chunks. Edges, which take the vertices in the chunk as destination, are also contained in the sub-graph. The set of edges in the sub-graph could be further divided by their source vertices. We put the edges with

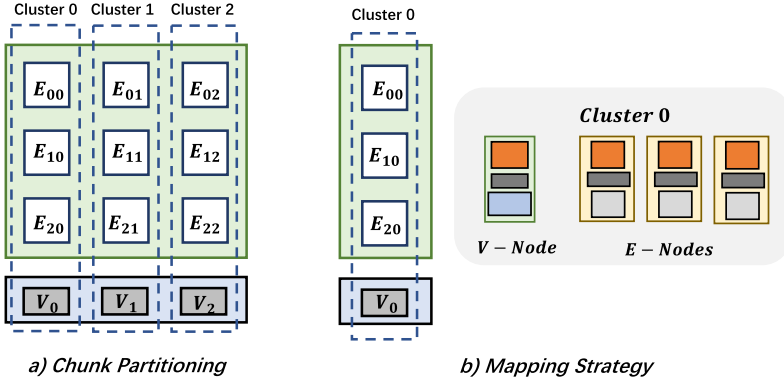


Fig. 2. Edge and vertex chunks partitioning and mapping strategy

source vertices in the same sub-graph into a block, named as edge chunk. And we assign a cluster of our architecture with a sub-graph, which contains a column of edge chunks for processing *ApplyEdge* on, together with the vertex chunk for handling *ApplyVertex* phase. In a cluster, we let an E-Node process a layer of neural network of *ApplyEdge*, and V-Nodes are responsible for handling *Scatter*, *Gather* and *ApplyVertex*.

As for most of natural graphs obey the power-law, the edges in each chunk may be diverse significantly, resulting in imbalanced workload for clusters. To handle this situation, we employ the Index Mapping Interval-Block Partition (IMIB) algorithm introduced in [9]. IMIB firstly removes the blank vertices, then hashes the vertices into different chunks using the modulo function. The time complexity of this algorithm is $O(m)$, where m denotes the size of edges.

4.2 Setup and Terminology

ApplyEdge is the most time-consuming state, which is normally composed of multiple layers of neural networks. To simplify discussion, we assign the computation task of each layer to an individual E-node. In the example of Fig. 2, the *ApplyEdge* phase employs a three-layer MLP. Thus, we have three E-nodes in a cluster for each layer computation.

Terminologies

- E_{ij} denotes the edge chunk located at the i -th row and the j -th column, and V_i means the i -th vertex chunk.
- L_i denotes the i -th E-Node, which handles the computation of i -th layer *ApplyEdge*.
- $S(E_{ij})$ denotes the output features of *Scatter*, which take V_i and V_j as input and process the operations related to E_{ij} .
- $L_t(E_{ij})$ denotes the output features of t -th layer on the edge chunk E_{ij} .

4.3 Dataflow Description

We divide the whole execution flow into multiple rounds. A round can be further divided into two sub-rounds. During the computation sub-round, input features and neural weights stored in their local buffer are fed into processing arrays to do the computation. During each communication sub-round, nodes forward their output features, and clusters exchange vertex chunks stored in temporary memories with each other in a circular manner, as shown in Fig. 3.

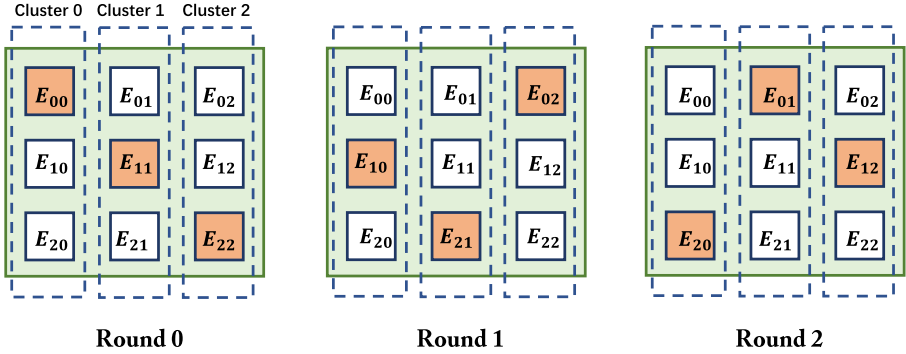


Fig. 3. Edge chunks being processed in each round

Initially, the V-Node in the cluster loads the i -th vertex chunk and copies it into both the stationary vertex memory and temporary vertex memory. In the following paragraphs, we will introduce the details in Round 0, Round 1 to illustrate how the dataflow works.

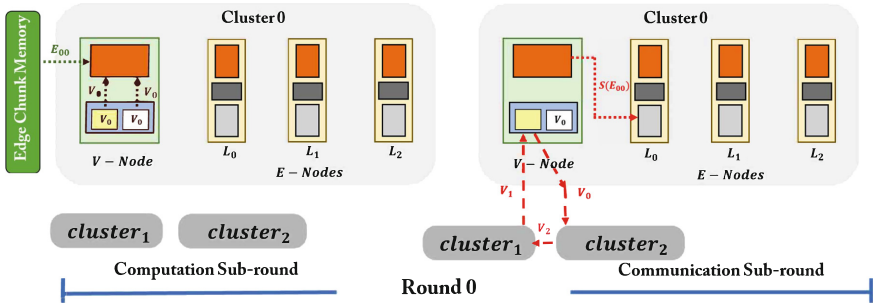


Fig. 4. GNN-PIM dataflow during Round 0

The dataflow during Round 0 is shown in Fig. 4. During the computation sub-round, clusters load edge chunks in the diagnose of the adjacent matrix of the graph, as shown in Fig. 3. Then V-Nodes feed data in edge and vertex chunks into processing cores, respectively. During the communication sub-round, the V-Node in cluster 0 forwards its output features $S(E_{00})$ to the edge node. Cluster 0 transmits its vertex chunk V_0 to its neighbor cluster 2. At the same time, it receives the vertex chunk V_1 from cluster 1.

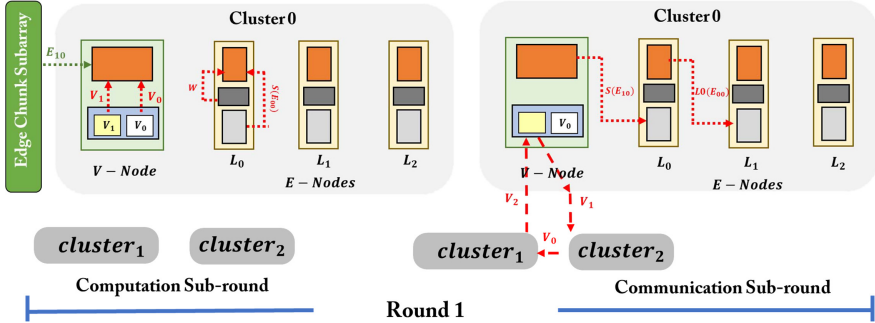


Fig. 5. GNN-PIM dataflow during Round 1

The dataflow during Round 1 is illustrated in Fig. 5. During the computation sub-round, node clusters shift the edge chunks they works on, loading new edge chunks from edge chunk memory, as Fig. 3 shows. Noted that the vertex chunks stored in stationary vertex memory and temporary vertex memory are exactly the destination vertices and source vertices of the new edge chunk respectively. E-node L_0 starts its computation in this round. During the communication sub-round, both V-Nodes and E-Nodes forward their outputs to their following nodes in the pipeline. At the same time, each cluster also forwards its vertex chunk accordingly.

In the subsequent rounds, GNN-PIM repeats this process until clusters receive the same data as the one in their stationary memory, which indicates that *ApplyEdge* and *Gather* finish. After that, V-Nodes continue to perform *ApplyVertex* on the aggregated data to generate new features. The *ApplyVertex* phase is not shown due to page limitation.

5 Interconnection Hierarchy

As reported by Ji et al. [12], data transmission may consume considerable time, even a number of times longer than computation does. For GNN inference processing, the transmission is much more complex by combining all the 4 different phases in SAGA. According to previous work [16], it is inefficient to employ basic interconnection topology such as bus or mesh for handling GNN’s transmission patterns. As a result, we develop a 2-layer hierarchical interconnection

network for handling not only local communication efficiently in the cluster but also global communication between clusters with low power and area overhead.

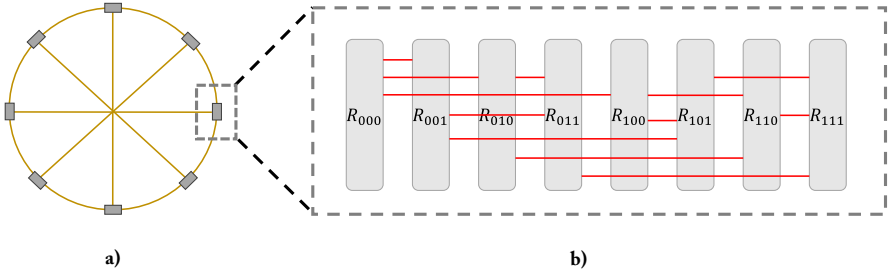


Fig. 6. Network hierarchy

Inter-cluster Interconnection Networks: We use the Octagon topology [15] for the global network, as shown in Fig. 6(a). Each block on the octagon is a node cluster. The $O(n)$ ring topology is cost effective, and can fit data movement pattern of the node cluster architecture well. The Octagon topology is constructed based on a ring network. It guarantees that there exist at most two hops of a transaction between two arbitrary routers in the same ring. As for cross-ring transaction, the maximum number of hops increases to $2n$, where n is the number of pass-by rings. Furthermore, the low complexity of global wiring demonstrates the great scalability of GNN-PIM’s networks.

Noted that the main cost of *Scatter* is to move data between clusters. To illustrate the superiority of our dataflow on this topology, we use the total hops in global network as the metric since traversing through global wires consumes much more time and energy than local ones. There are 8 clusters in total and they are connected by a single *Octagon* Ring. Without considering deadlock or non-optimal routing, the minimal global hops of preparing data for an E-Node is $1 \times \frac{3}{7} + 2 \times \frac{4}{7} = 1.57$ on average. And the number decreases to 1 by employing the optimized dataflow, as each transaction is only between two adjacent clusters. Furthermore, most nodes keep busy and no spatial broadcast is needed, which eliminates the bandwidth demand of global interconnection network. The storage overheads consist of the vertex of the graph and one edge chunk per cluster. These overheads can be ignored, compared with holding the whole edge data.

Intra-cluster Interconnection Networks: The topology employed to connect nodes in the same group is illustrated in Fig. 6(b), and each router is in charge of the data transmission related to a node. In this example, n routers are connected via the local network. Each router is connected to other $\log n$ routers, i.e. each router has the radix of $\log n$. Since each number with $h = \log n$ bits can change to another number with the same bit in h steps by inverting single bit per step, each transaction goes through at most $\log n$ hops from source to destination.

6 Evaluation

In this section, we first present the detailed evaluation setup. It is worth noticing that GNN-PIM is applicable for various memory technologies, and we choose ReRAM to prototype GNN-PIM in this work. Then we perform comparison between GNN-PIM and prior works in performance. We also present comprehensive analysis on power consumptions of GNN-PIM.

6.1 Benchmark

We use the datasets mentioned in Neugraph [20] to evaluate GNN-PIM’s performance and power consumptions. The vertex number, edge number, feature size, and model size of these real-world graphs are summarized in Table 1.

Table 1. Summary of real-world graphs

Dataset	Vertex	Edge	Feature	Storage
pubmed	19.7K	108.4K	500	500 KB
blog	10.3K	668.0K	128	2.6 MB
redditsmall	58.2K	1.4M	300	5.8 MB
redditfull	2.4M	705.9M	300	2883 MB
enwiki	3.6M	276.1M	300	1118 MB
amazon	8.6M	231.6M	96	960 MB

6.2 Methodology

Baseline: We employ a state-of-the-art design, PRIME [8], as our baseline. We adapt the size of PRIME circuits and scale the power and performance reported in that paper for a fair comparison. For simplicity, we treat its interconnection topology as a bus.

GNN-PIM Configuration: We set the overall size of ReRAM to 16 GB, and it is divided into 8 clusters with 2 GB each. There are 32 nodes in each cluster, connected by local networks. We statically assign 4 of them as V-Nodes while the other 28 nodes are N-Nodes. For processing elements, we adapt the same configurations used in PRIME [8]. Resolution of ADC and DAC is 5 bits and 2 bits, respectively. The power and area of the circuit are modeled based on ISAAC [24]. The HRS/LRS resistances are 25 M Ω /50 K Ω , and read/write voltages are 0.7 V/2 V. The latency and energy cost of read/write are 29.31 ns/50.88 ns and 1.08 pJ/3.91 nJ, respectively. The on-chip network design adapts Booksim [13] to simulate the latency, and employ the model proposed in ORION [14] to simulate power consumption and area overhead of the NoC. Both global network and local network work on 1 GHz. 4 nodes share a physical router with 1 global channel and 7 local channels. To fully utilize the bandwidth of local networks, we place V-Nodes of a cluster in different physical routers.

6.3 Performance Results

The time consumption normalized to PRIME is illustrated in Fig. 7. The performance of PRIME is obtained by applying the dataflow generated by SAGA framework directly on its circuits. PRIME’s primitive architecture leads to intensive competition and conflicts while using bus for data communication. As a result, the majority of PEs stay idle waiting for the data.

Therefore, the performance improvement of GNN-PIM comes from two folds. On one hand, hierarchy interconnection as well as optimized dataflow cooperate to reduce data transmission latency. On the other hand, parallel broadcast mechanism delivers data to multiple processing elements simultaneously, enabling more PEs working in parallel.

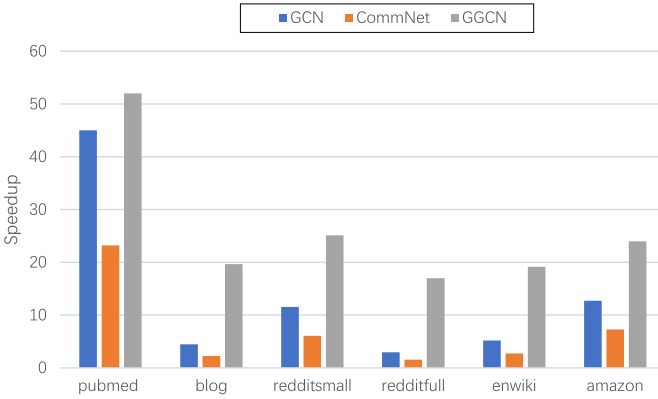


Fig. 7. Speedup over mapping GNNs directly on PRIME

Figure 8 illustrates the breakdown of power consumption, in which we choose GGCN as the benchmark. The power consumption of NoC could be heavy when



Fig. 8. Breakdown of power consumption

the graph is sparse or configured with short feature vectors. There are considerable redundant data transmission brought by edge-center programming model when the graph has high sparsity. Meanwhile, the feature size F will influence the computation with the factor of F^2 for matrix-vector multiplications, while for communication the factor is just F .

7 Conclusions

In this paper, we propose GNN-PIM, a PIM architecture for processing GNN inference. GNN-PIM employs a PIM-based hierarchical architecture with high throughput and efficiency. Besides, we perform customized optimizations on the dataflow, and propose a hierarchical NoC design to fully utilize the improvements brought by the optimized dataflow. Experimental results show that GNN-PIM achieves up to 52x speedup compared with prior designs.

Acknowledgement. This work is supported by National Key Research and Development Project of China (Grant No. 2018YFB1003304) and Beijing Academy of Artificial Intelligence (BAAI).

References

1. Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., Das, R.: Compute caches. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 481–492. IEEE (2017)
2. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015, pp. 105–117 (2015). <https://doi.org/10.1145/2749469.2750386>
3. Angizi, S., He, Z., Fan, D.: PIMA-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6 (2018)
4. Angizi, S., He, Z., Rakin, A.S., Fan, D.: CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator. In: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6 (2018)
5. Angizi, S., Sun, J., Zhang, W., Fan, D.: Aligns: a processing-in-memory accelerator for DNA short read alignment leveraging SOT-MRAM. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2019)
6. Asghari-Moghaddam, H., Son, Y.H., Ahn, J.H., Kim, N.S.: Chameleon: versatile and practical near-DRAM acceleration architecture for large memory systems. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–13. IEEE (2016)
7. Boroumand, A., et al.: CoNDA: efficient cache coherence support for near-data accelerators. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 629–642 (2019)
8. Chi, P., et al.: PRIME: a novel processing-in-memory architecture for neural network computation in ReRam-based main memory. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, 18–22 June 2016, pp. 27–39 (2016). <https://doi.org/10.1109/ISCA.2016.13>

9. Dai, G., et al.: GraphH: a processing-in-memory architecture for large-scale graph processing. *IEEE Trans. CAD Integr. Circ. Syst.* **38**(4), 640–653 (2019). <https://doi.org/10.1109/TCAD.2018.2821565>
10. Farmahini-Farahani, A., Ahn, J.H., Morrow, K., Kim, N.S.: NDA: near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 283–295. IEEE (2015)
11. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017*, pp. 1024–1034 (2017). <http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs>
12. Ji, Y., et al.: FPSA: a full system stack solution for reconfigurable ReRam-based NN accelerator architecture. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, 13–17 April 2019*, pp. 733–747 (2019). <https://doi.org/10.1145/3297858.3304048>
13. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21–23 April 2013*, pp. 86–96 (2013). <https://doi.org/10.1109/ISPASS.2013.6557149>
14. Kahng, A.B., Li, B., Peh, L., Samadi, K.: ORION 2.0: a power-area simulator for interconnection networks. *IEEE Trans. Very Large Scale Integr. Syst.* **20**(1), 191–196 (2012). <https://doi.org/10.1109/TVLSI.2010.2091686>
15. Karim, F., Nguyen, A., Dey, S.: An interconnect architecture for networking systems on chips. *IEEE Micro* **22**(5), 36–45 (2002)
16. Kwon, H., Samajdar, A., Krishna, T.: Rethinking NoCs for spatial neural network accelerators. In: *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2017, Seoul, Republic of Korea, 19–20 October 2017*, pp. 19:1–19:8 (2017). <https://doi.org/10.1145/3130218.3130230>
17. Li, S., Niu, D., Malladi, K.T., Zheng, H., Brennan, B., Xie, Y.: DRISA: a DRAM-based reconfigurable in-situ accelerator. In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 288–301. IEEE (2017)
18. Li, S., Xu, C., Zou, Q., Zhao, J., Lu, Y., Xie, Y.: Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6 (2016)
19. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016, Conference Track Proceedings* (2016). <http://arxiv.org/abs/1511.05493>
20. Ma, L., et al.: Neugraph: parallel deep neural network computation on large graphs. In: *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, 10–12 July 2019*, pp. 443–458 (2019). <https://www.usenix.org/conference/atc19/presentation/ma>
21. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010*, pp. 135–146 (2010). <https://doi.org/10.1145/1807167.1807184>
22. Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., Eliassi-Rad, T.: Collective classification in network data. *AI Mag.* **29**(3), 93–106 (2008). <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2157>

23. Seshadri, V., et al.: Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 273–287. IEEE (2017)
24. Shafiee, A., et al.: ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, 18–22 June 2016, pp. 14–26 (2016). <https://doi.org/10.1109/ISCA.2016.12>
25. Singh, G., et al.: NAPEL: near-memory computing application performance prediction via ensemble learning. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2019)
26. Song, L., Qian, X., Li, H., Chen, Y.: Pipelayer: a pipelined ReRam-based accelerator for deep learning. In: 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, 4–8 February 2017, pp. 541–552 (2017). <https://doi.org/10.1109/HPCA.2017.55>
27. Song, L., Zhuo, Y., Qian, X., Li, H.H., Chen, Y.: GraphR: accelerating graph processing using ReRam. In: IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, 24–28 February 2018, pp. 531–543 (2018). <https://doi.org/10.1109/HPCA.2018.00052>
28. Tang, L., Liu, H.: Relational learning via latent social dimensions. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, 28 June–1 July 2009, pp. 817–826 (2009). <https://doi.org/10.1145/1557019.1557109>
29. Xie, L., Du Nguyen, H.A., Taouil, M., Hamdioui, S., Bertels, K.: Fast boolean logic mapped on memristor crossbar. In: 2015 33rd IEEE International Conference on Computer Design (ICCD), pp. 335–342. IEEE (2015)
30. Yan, M., et al.: HyGCN: a GCN accelerator with hybrid architecture. CoRR abs/2001.02514 (2020). <http://arxiv.org/abs/2001.02514>
31. Yu, J., Du Nguyen, H.A., Xie, L., Taouil, M., Hamdioui, S.: Memristive devices for computation-in-memory. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1646–1651. IEEE (2018)
32. Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Sun, M.: Graph neural networks: a review of methods and applications. CoRR abs/1812.08434 (2018). <http://arxiv.org/abs/1812.08434>



A Software-Hardware Co-exploration Framework for Optimizing Communication in Neuromorphic Processor

Shiyong Wang, Lei Wang^(✉), Ziyang Kang, Lianhua Qu, Shiming Li, and Jinshu Su

School of Computer, National University of Defense Technology, Changsha, China
{wangshiyong18, leiwang, kangziyang14, qulianhua14, lishiming15, sjs}@nudt.edu.cn

Abstract. Spiking neural networks (SNN) has been widely used to solve complex tasks such as pattern recognition, image classification and so on. The neuromorphic processors which use SNN to perform computation have been proved to be powerful and energy-efficient. These processors generally use Network-on-Chip (NoC) as the interconnect structure between neuromorphic cores. However, the connections between neurons in SNN are very dense. When a neuron fire, it will generate a large number of data packets. This will result in congestion and increase the packet transmission latency dramatically in NoC.

In this paper, we proposed a software-hardware co-exploration framework to alleviate this problem. This framework consists of three parts: software simulation, packet extraction&mapping, and hardware evaluation. At the software level, we can explore the impact of packet loss on the classification accuracy of different applications. At the hardware level, we can explore the impact of packet loss on transmission latency and power consumption in NoC. Experimental results show that when the neuromorphic processor runs MNIST handwritten digit recognition application, the communication delay can be reduced by 11%, the power consumption can be reduced by 5.3%, and the classification accuracy can reach 80.75% (2% higher than the original accuracy). When running FSDD speech recognition application, the communication delay can be reduced by 22%, the power consumption can be reduced by 2.2%, and the classification accuracy can reach 78.5% (1% higher than the original accuracy).

Keywords: Neuromorphic processor · Communication optimization · Network-on-Chip · Reservoir computing

1 Introduction

Neuromorphic computing is an important branch of artificial intelligence. The concept of Neuromorphic computing was proposed by Carver Mead [10], which refers to the use of large-scale integrated circuit systems to simulate the neurobiological structure existing in the nervous system to complete the calculation of large-scale

neural networks. Recently, some excellent neuromorphic processors have emerged, such as SpiNNaker [6], BrianScales [18], TrueNorth [1], DYNAPs [16], Loihi [4], and Tianjic [17]. All of these neuromorphic processors contain thousands of neurons and synaptic connections. For better scalability and parallelism, they mostly use Network-on-Chip (NoC) to make connections between neurons.

In general, Spiking neural network (SNN) is used in the neuromorphic processor. SNN is the third generation artificial neural network (ANN) [15] which is originally inspired by brain science. It has been proved to be a powerful and energy-efficient computation model and has been applied to complex tasks such as pattern recognition [20], image classification [9], natural language processing [5], etc. Many SNN models can be implemented on the neuromorphic processor, such as multilayer perceptron (MLP), spiking convolutional neural network (SCNN) [8], reservoir computing (RC) [21], etc. Compared with other models, RC has more advantages in terms of application scope and implementation complexity.

Unlike other SNN models, neurons in the RC model can be connected not only to other neurons but also to themselves. This characteristic makes the connections denser among the neurons. Figure 1 shows the distribution of the number of neuron connections in an RC network. The horizontal axis represents the number of connections. The vertical axis refers to the number of neurons corresponding to a specific number of connections. We can find that all neurons are connected to at least one-third of the neurons in the RC.

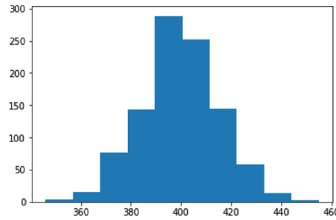


Fig. 1. Connections distribution in an RC network: 1000 neurons in total, 800 excitory neurons, 200 inhibit neurons. $P_{ee} = 0.40$, $P_{ei} = 0.40$, $P_{ie} = 0.50$.

In SNN, one pre-neuron cloud connects a large number of post-neurons. When the pre-neurons membrane voltage exceeds its threshold and fire, the same count of spike will be generated. Therefore, when the RC network is running in a neuromorphic processor, a large number of data packets will be generated at the same time. This will cause packet congestion in NoC and increase packet transmission latency. Figure 2 is a simple running process in a neuromorphic processor. Neuron *A* connects three neurons *B*, *C*, and *D*. They are mapped to different neuromorphic cores. At time-step *t*, the membrane voltage of *A* exceeds its threshold. Then, neuron *A* generates a spike, and send it to all the post-neurons connected to it. Because the router cannot transmit all packets immediately, the final packet has a greater transmission latency than other packets.

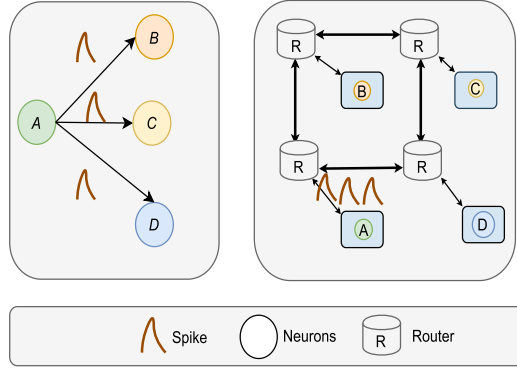


Fig. 2. The running process of SNN in neuromorphic processor. Neuron *A* connects three neurons *B*, *C*, and *D*. At time-step *t*, neuron *A* generates a spike.

When a large number of packets need to be transmitted at the same time, it will cause NoC congestion. At the same time, through experiments, we find that when a small number of data packets are discarded, it doesn't affect the classification accuracy of the software. Based on the above problems and phenomena, in this paper, we proposed a software-hardware co-exploration framework to optimize the communication in a neuromorphic processor. This framework consists of three parts: software simulation, packet extraction & mapping, and hardware evaluation. According to it, we can optimize the communication in the neuromorphic processor while ensuring the accuracy of software recognition. To this end, we make the following contributions:

- 1) We explored the impact of packet loss on the accuracy of SNN classification. It is found that a low spike loss rate doesn't affect the classification accuracy of SNN. In some cases, the classification accuracy is even higher than the original accuracy.
- 2) We propose a software-hardware co-exploration framework for optimizing communication in a neuromorphic processor. It can alleviate the congestion in NoC and reduce the packet transmission latency without reducing the accuracy of SNN classification.

After optimization, when the neuromorphic processor runs MNIST handwritten digit recognition application, the communication delay can be reduced by 11%, the power consumption can be reduced by 5.3%, and the classification accuracy can reach 80.75% (2% higher than the original accuracy). When running FSDD speech recognition application, the communication delay can be reduced by 22%, the power consumption can be reduced by 2.2%, and the classification accuracy can reach 78.5% (1% higher than the original accuracy).

2 Background and Related Work

2.1 RC Model

Reservoir calculation (RC) is a type of RNN model, which is mainly composed of an input layer, reservoir layer, and output layer, as shown in the Fig. 3. The reservoir layer consists of a certain number of excitatory spiking neurons and inhibitory spiking neurons. According to the complexity of the task, the number of two types of neurons and the connection probability between them are different. W_{In} , W_R , and W_{Out} represent the weights of the input layer, reservoir layer, and output layer, respectively. After the reservoir layer receives the input spike train and runs for a certain period, it will generate an RC state (usually the membrane voltage of the neurons in the RC). The output layer records the state of neurons in the reservoir layer.

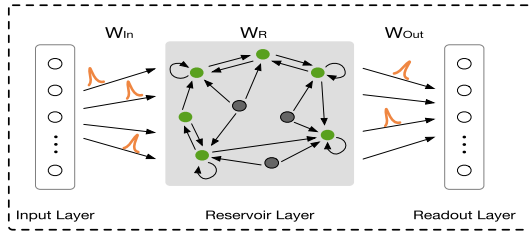


Fig. 3. The architecture of RC.

2.2 Network-on-Chip

With the advent of multi-core processors, the design interconnect of System on Chip (SoC) faces many challenges. SoCs typically use a Network on Chip (NoC) [2, 12] solution, with various NoC topologies and router architectures, and provide low power and high quality of service (QoS) designs. NoC has many components, such as topology, routing algorithms, and router microstructure design.

TrueNorth [1] is a neuromorphic chip developed by IBM. In TrueNorth, its time-step is 1 ms. Operation in each time-step is divided into two phases: In the first phase, data packets will be routed through the Router. When the data packet reaches the corresponding core, it will change the membrane voltage of the corresponding neuron. In the second phase, all cores will receive a synchronizing signal with a period of 1 ms. Once the synchronizing signal is received, all neurons need to check whether their membrane voltage exceeds the threshold. If the threshold is exceeded, the neuron will send a data packet to the network.

TrueNorth uses a global synchronous clock to synchronize each time-step, so the size of the global clock must consider the worst case in the entire chip. However, not all packets have a large transmission latency. So its synchronization method will reduce the efficiency of the hardware.

3 The Software-Hardware Co-exploration Framework

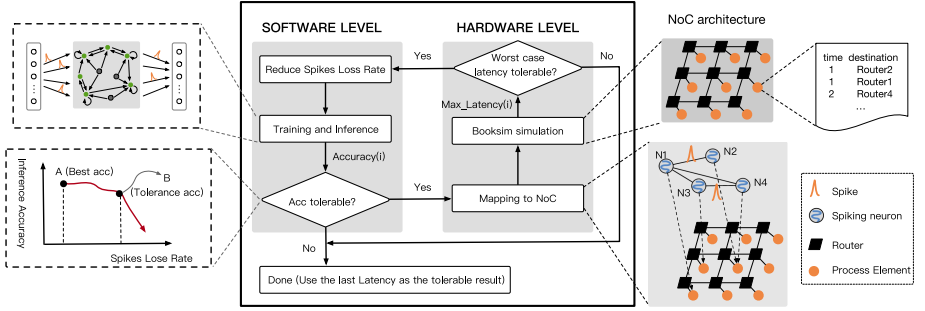


Fig. 4. Software hardware co-exploration framework.

3.1 Real-Time Definition

Define that a synchronization period in neuromorphic processor consumes L cycles. When an application is running in a neuromorphic processor, if the data packets generated by it meet the following conditions, we think that the application meets the real-time requirements of the neuromorphic processor:

$$\text{Max}(\text{Latency}_i) < L \quad (1)$$

Among them, i belongs to $[0, N - 1]$, N is the total amount of data packets generated by the application in the inference process. In other words, the transmission latency of all packets is less than the length of the synchronization cycle.

3.2 Framework

Our proposed framework is shown in Fig. 4. It mainly composed three parts:

Software Level Simulation. We use the SNN simulator to run the RC network. When neurons communicate, we drop packets with different probabilities and explore the impact of different packet loss rates on software classification accuracy.

Many SNN simulators can be used, such as Brian2, CARLsim [3], Nest [7], etc. They can be used to simulate the behavior of neurons and simulate the operation of SNN. At the same time, log files can be extracted during the running process. During initialization, we can determine the connection probability and the weight of the connection in the network. After initialization, we extract the connection relationship.

We explore the impact of packet loss on the accuracy of SNN classification as follows: 1) Generate application input spike. 2) Train the readout layer, perform

classification, and finally test multiple test-sets to get the accuracy of the classification. 3) Change the different spike loss rates to get the relationship between classification accuracy and spike loss rate. By dropping part of the data packets, the communication delay in the neuromorphic processor can be reduced. Synchronization in the neuromorphic processor must satisfy the worst communication situation. Therefore, this can reduce the number of clock cycles required for global synchronization.

Trace Extraction and Mapping. The spike trace can be recorded when *Brian2* is simulating. Each trace records spikes from one neuron to another neuron. The format of each trace in the trace file is:

$$[Source\ Neuron\ ID, Destination\ Neuron\ ID, time-step]$$

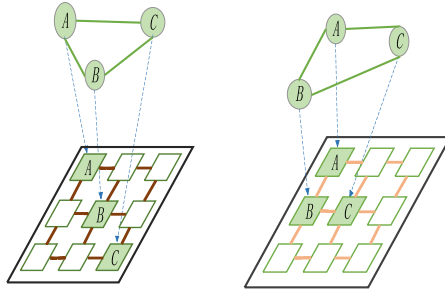


Fig. 5. Two different mapping methods.

Mapping is a process that map neurons in SNN to cores of neuromorphic processor. Figure 5 shows two different mapping methods [14].

We use NoC simulator to simulate the communication between the cores in the neuromorphic processor when the SNN network is running.

Hardware Level Evaluation. At the hardware level, we simulate communication between neuromorphic cores through a NoC simulator. The detailed hardware evaluation process is shown in Fig. 6. As mentioned before, the parameters of NoC include topology, routing algorithm, router micro-architecture, etc. In this work, we use a 2D-Mesh network structure and a dimension-order-first routing algorithm to route packets.

As shown in Fig. 6, NoC configuration file and trace files are inputs for hardware evaluation, latency and power consumption are outputs. Throughput refers to the number of data packets transmitted by the NoC within a specified time. In neuromorphic processors, each spike is a data packet. Throughput is generally defined by the following formula (2):

$$Throughput = \frac{(Total\ Spike\ packet\ finished) \times Packet_{Length}}{(Numbers\ of\ Router) \times Total_{time}} \quad (2)$$

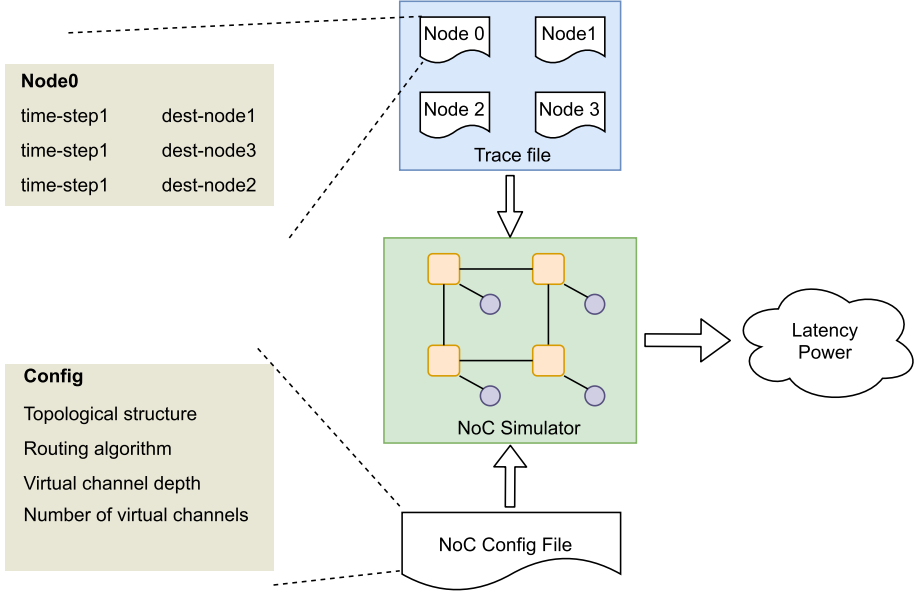


Fig. 6. Hardware level simulation.

The $Packet_{Length}$ is measured in flits. Each packet can be represented by a spike. So we defined the $Packet_{Length} = 1$.

Let P be the total number of messages reaching their destination and let L_i be the latency of each message i , where i ranges from 1 to F , as shown in formula (3):

$$L_i = \text{receiving time of } P_i - \text{generating time of } P_i \quad (3)$$

Max transport latency is computed as shown in formula (4):

$$L_{Transport\ latency} = Max(L_i), 1 \leq i \leq F \quad (4)$$

3.3 Framework Workflow

The framework workflow is shown in Algorithm 1. We can tolerate a 1% reduction in classification accuracy at most. If the real-time requirements are not met, the packet loss rate will increase by 5% at a time. If the real-time requirement can be met, the packet loss rate and classification accuracy will be output. If the requirement is not met, the framework will explain the situation and exit.

Algorithm 1. Framework workflow

Input: *Real_time_Latency*
Output: *spike_loss_rate, Accu*

- 1: *Real_time_Latency*//Maximum packets transmission latency that meet real-time requirements;
- 2: *spike_loss_rate* = 0
- 3: *Accuracy(i)*//calculation classification accuracy when the spike loss rate is *i*;
- 4: *Tolerance_acc* = *Accuracy(0)* - 0.01//The minimum value of classification accuracy we can tolerate;
- 5: *Max_Latency(i)*//Maximum packets transmission latency when the packet discarding rate is *i*;
- 6: *Latency* = *Max_Latency(0)*
- 7: **while** *Latency* > *Real_time_Latency* **do**
- 8: *spike_loss_rate*+ = 0.05
- 9: *Accu* = *Accuracy(spike_loss_rate)*
- 10: **if** *Accu* > *Tolerance_acc* **then**
- 11: *Latency* = *Max_Latency(spike_loss_rate)*
- 12: **if** *Latency* <= *Real_time_Latency* **then**
- 13: return *spike_loss_rate, Accuracy*
- 14: **else**
- 15: continue
- 16: **end if**
- 17: **else**
- 18: Classification accuracy reduced too much.
- 19: Unable to meet the real-time requirement.
- 20: break
- 21: **end if**
- 22: **end while**

4 Experiment and Analysis

4.1 Experiment Setup

To verify the optimization effect of our framework, we set up the experiment as follows:

We use the *Brian2* simulator [19] to simulate the running of the RC network. Software-level simulation includes two steps: (1) Generate spikes from the data set. The content of the original data set cannot be directly used as the input of the SNN. Therefore, the original data set needs to be extracted and transformed into a spike train that can be understood by the SNN; (2) RC simulation and classification. We input the spike obtained from (1) into the RC network. After a period of time, readout layer reads the state of the RC neurons to train and classify. Through multiple tests, a classification accuracy can be obtained.

We use the clock-accurate NoC simulator *Booksim2* [11] to simulate the communication process of RC networks in NoC. We have modified *Booksim* so that it can read spike communication trace. The NoC topology in this work is

8×8 . The routing algorithm is x-y routing. The NoC has 8 virtual channels, and the virtual channel depth is 1. Detailed configuration of NoC is shown in Table 1.

We tested two different applications, one is MNIST handwritten digit recognition and the other is FSDD speech recognition.

Table 1. NoC Config.

topology	mesh
routing_function	xy
vc_allocator	islip
arb_type	round_robin
priority	age
num_vcs	8
vc_buff_size	1

4.2 Result of MNIST Dataset

MNIST [13] is a frame-based dataset, which is used for handwritten digits recognition.

Optimization Effect of Framework. Figure 7 shows optimization results of MNIST handwritten digit recognition application. Our target maximum transmission latency is 400 cycles. After several rounds of exploration, when the packet loss rate is 10%, the maximum transmission latency is reduced from the original 436 cycles to 392 cycles. It meets the real-time requirements. After optimization, the power consumption is reduced by 5.3%. At the same time, the classification accuracy didn't decline but increased from 78.75% to 80.75%.

Impact of Packet Loss on Performance of RC Network. Figure 8 shows the relationship between the packet loss rate and software classification accuracy. When the packet loss rate is in the range of 0%–10%, the classification accuracy is almost unchanged. When the packet loss rate exceeds 20%, the classification accuracy decreases significantly.

Impact of Packet Loss on Data Transmission Latency in NoC. Figure 9 shows the relationship between the packet loss rate and the maximum transmission latency in NoC. We can find that as the packet loss rate increases, the maximum transmission latency decreases significantly. Reduced transmission latency can increase hardware efficiency.

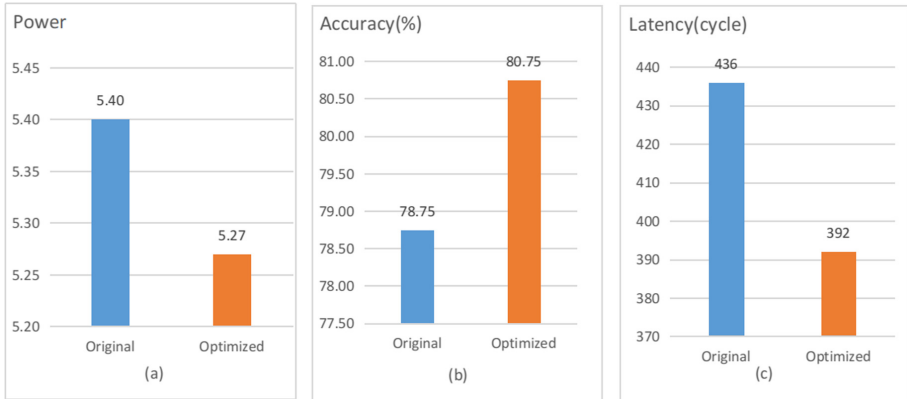


Fig. 7. Optimization results of MNIST handwritten digit recognition application. *Original* refers to the result without optimization. *Optimized* refers to the optimized result. (a) Power. (b) Accuracy. (c) Latency.

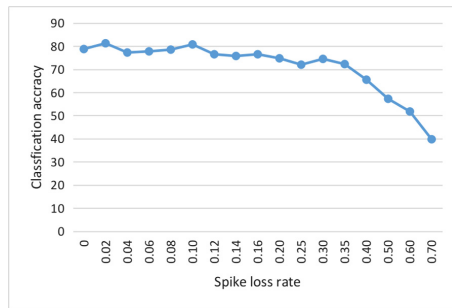


Fig. 8. Impact of packet loss on classification accuracy of MNIST handwritten digit recognition application.

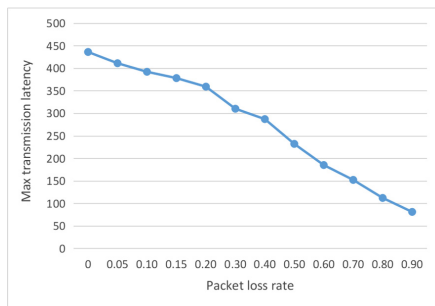


Fig. 9. Impact of packet loss on data transmission latency in MNIST handwritten digit recognition application.

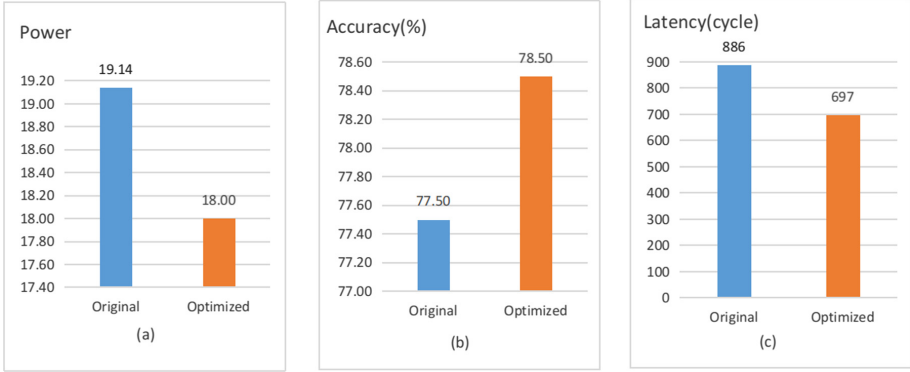


Fig. 10. Optimization results of FSDD speech recognition application. *Original* refers to the result without optimization. *Optimized* refers to the optimized result. (a) Power. (b) Accuracy. (c) Latency.

4.3 Result of FSDD Dataset

FSDD is a simple audio and speech dataset consisting of recordings of spoken digits at 8 kHz.

Optimization Effect of Framework. Figure 10 shows optimization results of FSDD speech recognition applications. For FSDD speech recognition applications, our target maximum transmission latency is 700 cycles. After several rounds of exploration, when the packet loss rate is 19%, the maximum transmission latency is reduced from the original 886 cycles to 697 cycles. It meets the real-time requirements. After optimization, the power consumption is reduced by 2.2%. At the same time, the classification rate didn't decline but increased by 1.0% from 77.5% to 78.5%.

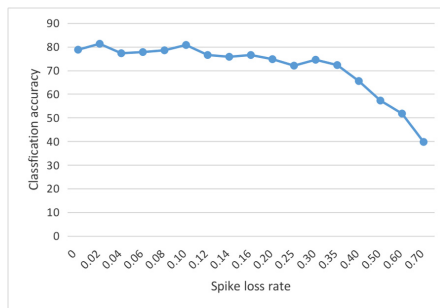


Fig. 11. Impact of packet loss on classification accuracy of FSDD speech recognition application.

Impact of Packet Loss on Performance of RC Network. As shown in Fig. 11, it shows the impact of packet loss on classification accuracy in FSDD speech recognition. When the packet loss rate is in the range of 0%–19%, the classification accuracy is almost unchanged. When the packet loss rate exceeds 19%, the classification accuracy decreases significantly. At the same time, with the packet loss rate is 1%, the classification accuracy is even better than the effect of no packet loss.

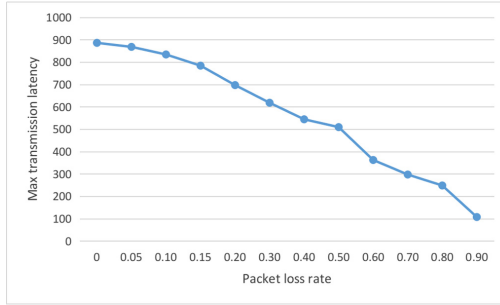


Fig. 12. Impact of packet loss on data transmission latency in FSDD speech recognition application.

Impact of Packet Loss on Packet Transmission Latency in NoC. Figure 12 shows the relationship between the packet loss rate and the maximum transmission latency in NoC for FSDD. Same as the result of MNIST handwriting recognition, we can find that as the packet loss rate increases, the maximum transmission latency decreases significantly.

4.4 Analysis

How Does Packet Loss Affect the Classification Accuracy of RC? As mentioned before, an RC network consists of three layers: the input layer, the reservoir layer, and the output layer. Once the reservoir layer is initialized, the connections between neurons and their corresponding weights no longer change. Training an RC network refers to training its readout layer. The input of the readout layer is the state of the neurons in the reservoir layer.

There are different ways to encode the state of a neuron, such as time encoding and frequency encoding. Frequency encoding means that the state of a neuron is represented according to the count of fire in the time window. Frequency encoding is used in this experiment. Packet loss adds “noise” to the reservoir layer. The inputs for training and testing are the states of the reservoir neurons that contain “noise”. These “noises” are randomly generated with a specific probability.

When the packet loss rate is very small, the classifier trained by the dataset containing “noise” can classify normally. At the same time, because the packet loss behavior is random, the corresponding classification accuracy may fluctuate around the original classification accuracy.

When the packet loss rate is too high, the spiking frequency of neurons will decrease. If frequency coding is used, the state difference between neurons will become less obvious. At this time, it’s hard for classifier to make right decision. As a result, the accuracy of classification will descend significantly.

5 Conclusion

In this paper, we proposed a software-hardware co-exploration framework for optimizing communication in neuromorphic processor. This framework consists of three parts: software simulation, packet extraction & mapping, and hardware evaluation. Without reducing the accuracy of SNN classification, we can alleviate the congestion in neuromorphic processor and reduce the packet transmission latency.

The experiment result shows that after optimization, when the neuromorphic processor runs MNIST handwritten digit recognition application, the communication delay can be reduced by 11%, the power consumption can be reduced by 5.3%, and the classification accuracy can reach 80.75% (2% higher than the original accuracy). When running FSDD speech recognition application, the communication delay can be reduced by 22%, the power consumption can be reduced by 2.2%, and the classification accuracy can reach 78.5% (1% higher than the original accuracy).

References

1. Akopyan, F., et al.: Truenorth: design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* **34**(10), 1537–1557 (2015)
2. Beigné, E., Clermidy, F., Vivet, P., Clouard, A., Renaudin, M.: An asynchronous NOC architecture providing low latency service and its multi-level design framework. In: 11th IEEE International Symposium on Asynchronous Circuits and Systems, pp. 54–63. IEEE (2005)
3. Chou, T.S., et al.: CARLsim 4: an open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In: 2018 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2018)
4. Davies, M., et al.: Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* **38**(1), 82–99 (2018)
5. Diehl, P.U., Zarella, G., Cassidy, A., Pedroni, B.U., Neftci, E.: Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. In: 2016 IEEE International Conference on Rebooting Computing (ICRC), pp. 1–8. IEEE (2016)
6. Furber, S.B., et al.: Overview of the spinnaker system architecture. *IEEE Trans. Comput.* **62**(12), 2454–2467 (2012)

7. Gewaltig, M.O., Diesmann, M.: Nest (neural simulation tool). *Scholarpedia* **2**(4), 1430 (2007)
8. Guo, S., et al.: A systolic SNN inference accelerator and its co-optimized software framework. In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pp. 63–68 (2019)
9. Iakymchuk, T., Rosado-Muñoz, A., Guerrero-Martínez, J.F., Bataller-Mompeán, M., Francés-Víllora, J.V.: Simplified spiking neural network architecture and STDP learning algorithm applied to image classification. *EURASIP J. Image Video Process.* **2015**(1), 4 (2015). <https://doi.org/10.1186/s13640-015-0059-4>
10. Izhikevich, E.M.: Simple model of spiking neurons. *IEEE Trans. Neural Netw.* **14**(6), 1569–1572 (2003). <https://ieeexplore.ieee.org/document/1257420>
11. Jiang, N., Michelogiannakis, G., Becker, D., Towles, B., Dally, W.J.: *Booksim 2.0 user’s guide*. Stanford University (2010)
12. Kumar, S., et al.: A network on chip architecture and design methodology. In: *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design, ISVLSI 2002*, pp. 117–124. IEEE (2002)
13. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
14. Li, S., et al.: SNEAP: a fast and efficient toolchain for mapping large-scale spiking neural network onto NOC-based neuromorphic platform. In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (2020)
15. Maass, W.: Networks of spiking neurons: the third generation of neural network models. *Neural Netw.* **10**(9), 1659–1671 (1997)
16. Moradi, S., Qiao, N., Stefanini, F., Indiveri, G.: A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE Trans. Biomed. Circ. Syst.* **12**(1), 106–122 (2017)
17. Pei, J., et al.: Towards artificial general intelligence with hybrid Tianjic chip architecture. *Nature* **572**(7767), 106–111 (2019)
18. Schemmel, J., et al.: Live demonstration: a scaled-down version of the brainscales wafer-scale neuromorphic system. In: *2012 IEEE International Symposium on Circuits and Systems*, pp. 702–702. IEEE (2012)
19. Stimberg, M., Brette, R., Goodman, D.F.: Brian 2, an intuitive and efficient neural simulator. *Elife* **8**, e47314 (2019)
20. Wysocki, S.G., Benuskova, L., Kasabov, N.: Fast and adaptive network of spiking neurons for multi-view visual pattern recognition. *Neurocomputing* **71**(13–15), 2563–2575 (2008)
21. Young, A.R., Dean, M.E., Plank, J.S., Rose, G.S.: A review of spiking neuromorphic hardware communication systems. *IEEE Access* **7**, 135606–135620 (2019)



A CNN Hardware Accelerator in FPGA for Stacked Hourglass Network

Dongbao Liang, Jiale Xiao, Yangbin Yu, and Tao Su^(✉)

San Yat-Sen University, Guangzhou 510006, Guangdong, China
sutao@mail.sysu.edu.cn

Abstract. Staked hourglass network is a widely used deep neural network model for body pose estimation. The essence of this model can be roughly considered as a combination of Deep Convolutional Neural Networks (DCNNs) and cross-layer feature map fusion operations. FPGA gains its advantages in accelerating such a model because of the customizable data parallelism and high on-chip memory bandwidth. However, different with accelerating a bare DCNN model, stacked hourglass networks introduce implementation difficulty by presenting massive feature map fusion in a first-in-last-out manner. This feature introduces a larger challenge to the memory bandwidth utilization and control logic complexity on top of the already complicated DCNN data flow design. In this work, an FPGA accelerator is proposed as a pioneering effort on accelerating the stacked hourglass model. To achieve this goal, we propose an address mapping method to handle the upsample convolutional layers and a network mapper for scheduling the feature map fusion. A 125 MHz fully working demo on Xilinx XC7Z045 FPGA achieves a performance of 8.434 GOP/s with a power efficiency of 4.924 GOP/s/W. Our system is $296\times$ higher than the compared Arm Cortex-A9 CPU and $3.2\times$ higher power efficiency, measured by GOP/s/W, than the GPU implementation on Nvidia 1080Ti.

Keywords: Stacked hourglass network · Convolutional Neural Network · Pose estimation · Hardware accelerator · FPGA

1 Introduction

In deep Convolutional Neural Networks (DCNNs), deeper layers yield larger perception fields to the original images and, therefore, encapsulate higher-level feature information, which is opposite to the shallow (near-input) layers. In body pose estimation tasks, a common practice to accurately locate body key points is to combine the abstract global features with shallow local features [1]. Following such motivation, stacked hourglass network are commonly used in pose estimation tasks and achieves an unprecedented accuracy on the MPII Human Pose dataset with an average 90.9% percentage of detections [1]. However, this design of model poses new challenges to the system design complexity on FPGAs compared to solely accelerating a DCNN model for classification tasks [4–7].

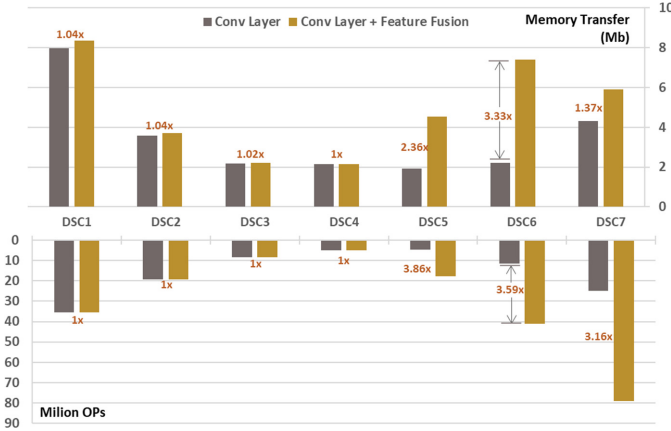


Fig. 1. Feature map fusion branches introduce up to 3.3 times more memory transfer and up to 3.59 times more operations to normal convolutional layers

In Fig. 1, DSC stands for depthwise separable convolution module [17] and DSC1-DSC7 are the layers in a single hourglass model (see model details in Sect. 2). The feature maps in the shallow layers are transferred between on- and off-chip for the feature fusion branches, which introduces up to 3.3 times more memory transfer and 3.59 times more computation compared to a simple cascaded convolutional layer structure. Such computation nature changes the workload from a compute-bound task into a memory-bound one [12, 19].

Central Processing Unit (CPU) suffers from its sequential execution nature when dealing with the parallel DCNN computation. General Purpose Graphic Processing Unit (GPGPU) is also a naturally fitted platform in accelerating DCNN applications, but the sweet point for GPGPUs normally requires a relatively large input batch size [11]. Additionally, the data synchronization mechanism in GPGPUs is not specially designed for neural network computation. Field-Programmable Gate Array (FPGA), on the other hand, gains its advantages in accelerating deep neural network tasks because of its customizable parallel logic and high power efficiency. There have been many implementations of DCNNs on FPGA in recent years [2–14]. Systems like NEURAghe [6] and SnowFlake [5] mainly target on DCNN classification task acceleration. As above-mentioned, DCNN classification models do not introduce extra computation and memory transfer from the feature fusion branches. It is hard to make straight and fair comparisons to these works. For example, if scales up a typical backbone DCNN accelerated in [5, 6] adding feature map fusion branches, the system performance reported in these works can massively drop due to the multi-scale feature fusion operations and newly introduced memory transfer. Although very few works exactly address the stacked hourglass network, its building block, depthwise separable convolution modules, are carefully considered in the system architecture designs in [10–12]. However, model redundancy reduction techniques are applied in these works to help realizing feasible FPGA mapping. These techniques, moreover, are orthogonal to our proposed system.

In most of the FPGA-based DCNN implementations, designers are benefited from configurable architecture to perform multiple functions of the algorithms and on-chip memories to overcome the limitation of the memory bandwidth. Works like [2] and [3] try to realize a general structure for all possible network structures with parameterized design, while other references [4] and [8] try to fit the CNN model into the embedded applications by dedicated memory arrangement and tiling strategies. Reference [9] and [10] explore the sparsity within the DCNN, using compression or pruning method to make fully use of the memory bandwidth. With the rapid development of the algorithms, new DCNN structures appears and reference [13] and [14] explore new mapping strategies for the networks with residual blocks like ResNet and Xception.

In this work, our new system architecture is designed for the stacked hourglass model with above difficulty handled by our proposed network mapper module and an address mapping method. The main contributions of this work are listed as below:

1. Our work is pioneering effort in accelerating the commonly used stacked hourglass network model on FPGAs. Our system achieves $296\times$ higher than the compared CPU and $3.2\times$ higher power efficiency than the examined GPU implementation.
2. To implement the logic for the feature map fusion branches, we propose a network mapper module for efficiently managing the storage and fetching of long-distant residual results. In addition, the network mapper also converts the stacked hourglass network into instructions for the accelerator automatically.
3. “Channel-first” data arrangement is proposed to enhanced the performance of 1×1 convolutions which are heavily used in depthwise separable convolutions.

This paper is organized as follows. Section 2 provides the background of stacked hourglass network and the separable depthwise convolution. Section 3 describes the architecture of the accelerator, including the processing engine and the organization of the on-chip memory. The dedicated design of control signals for the accelerator will be described in Sect. 4. Section 5 gives the experimental results on the performance of the accelerator, also provides a demo of the accelerator using on the real-time pose estimation application. The summary will be given in Sect. 6.

2 Background

2.1 Stacked Hourglass Network

Stacked hourglass network was first introduced in [1] to improve the accuracy of pose estimation task. Figure 2 gives us an overview of the architecture of an hourglass module and the stacked hourglass network. In the network, the input feature map is firstly scaled down to the intermediate with very low resolution but much more channels, then scaled up to the original size again. Scale down is done by the stride- n ($n > 1$) convolutions or pooling and the upsample performs the reverse operation. It’s worth noticing that feature maps are combined across multiple resolutions by doing summations within the hourglass module, named by its shape. Residual module is also used heavily in the hourglass module and filters greater than 3×3 are abandoned here, which make it easier for

the hardware design. The residual module sometimes is replaced by depthwise separable convolution to reduce the complexity of the network and the number of parameters further, which will be introduced in the next chapter. After the intermediate supervision process for the output of the hourglass module is included, hourglass modules can be stacked up successively to create a deeper network.

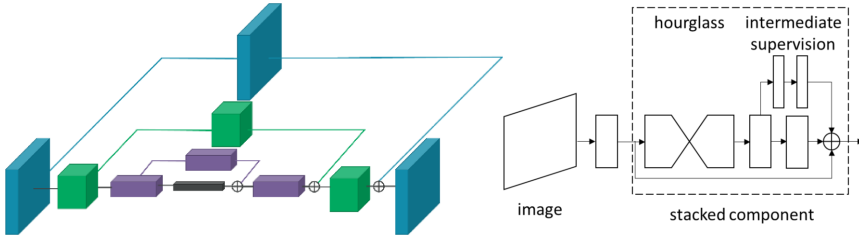


Fig. 2. An illustration of a single “hourglass” module (left), and the stacked hourglass network (right).

2.2 Depthwise Separable Convolution

Depthwise separable convolution was first introduced in [16]. In this kind of convolution, a standard convolution is split into two steps. Depthwise convolution firstly extracts features on the input feature map of different channels separately, and then pointwise convolution follows up to combine all the results in different channels with 1×1 convolution. Differences between standard convolution, depthwise convolution and pointwise convolution are illustrated in Fig. 3.

Depthwise separable convolution is proved to have much less parameters and arithmetic operations to achieve comparable accuracy for popular CNN networks [15]. A simple mathematical proof is shown here. Considering an $D \times D \times M$ input feature map produces an $D \times D \times N$ output feature map after a convolution operation. If standard convolution is chosen, the number of parameters P for the kernel with size of $K \times K$ is

$$P_{SC} = K \times K \times M \times N \quad (1)$$

And the computational cost O of the standard convolution is

$$O_{SC} = D \times D \times K \times K \times M \times N \quad (2)$$

However, if depthwise separable convolution is chosen, the corresponding number of parameters and the computational cost is

$$P_{DSC} = K \times K \times M + M \times N \quad (3)$$

$$O_{DSC} = D \times D \times K \times K \times M + D \times D \times M \times N \quad (4)$$

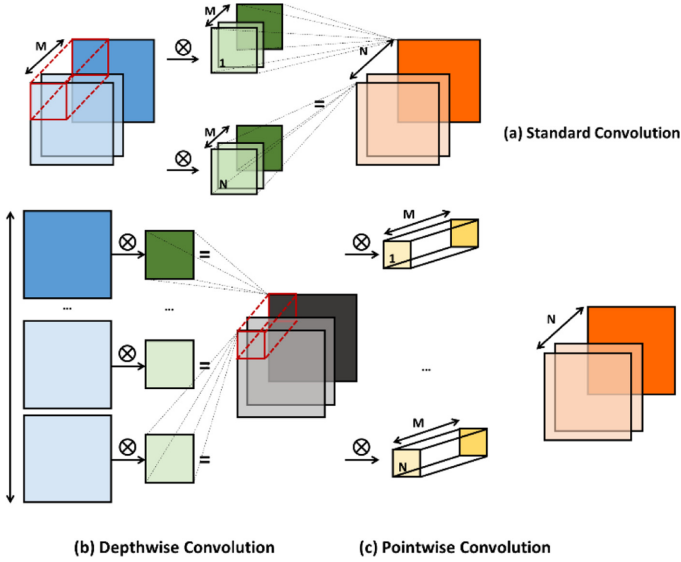


Fig. 3. Comparison of different kinds of convolutions.

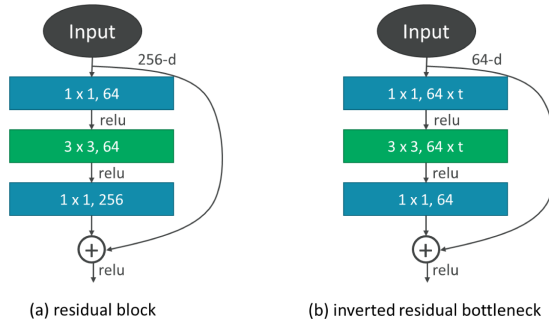


Fig. 4. Different kinds of residual blocks. Note that t ($t > 1$) in (b) is the expansion factor.

Thus, the reduction of the parameters and the computational cost comparing to the standard convolution is

$$F_P = \frac{K \times K \times M + M \times N}{K \times K \times M \times N} = \frac{1}{N} + \frac{1}{K^2} \quad (5)$$

$$F_O = \frac{D \times D \times K \times K \times M + D \times D \times M \times N}{D \times D \times K \times K \times M \times N} = \frac{1}{N} + \frac{1}{K^2} \quad (6)$$

A popular choice of K is 3 so the common reduction in number of parameters and the computational cost is 8 to 9 times.

Depthwise separable convolution is successfully applied to the usual object detection and classification tasks in MobileNetV1 [15], and the successor MobileNetV2 [18]. In MobileNetV2, performance is further improved by the new operation called bottleneck, also get its name from the shape. Within the structure, one more 1×1 convolution

is placed before the depthwise convolution and the feature map fusion connects two bottlenecks from bottom to top. The inverted residual bottleneck layers (differ from the common residual block in [17]) is proved to be memory efficient evidently for feature maps with less channels in the bottleneck (Fig. 4). This structure is also widely used in stacked hourglass network acting as a substitution of residual modules.

3 Hardware Design

In this chapter, we present the architecture of the hardware accelerator for running stacked hourglass network in the inference phase. The acceleration lies in the dedicated data path design and memory management.

3.1 Overall Architecture

The block diagram in Fig. 5 shows the overview of the whole architecture of the accelerator. Within the accelerator, a control module receives the control instructions from the host CPU and transform them into internal control signals for the other module in the accelerator. Processing engine (PE) array is responsible for all calculations in stacked hourglass network. Buffer module moves input data and weight from external memory to on-chip buffer, also move the calculated results came out from PE array back to DRAM. Redistribution module is placed between PE array and buffer module for order rearrangement for input data and output results, which will be introduced in detail in Sect. 3.2.

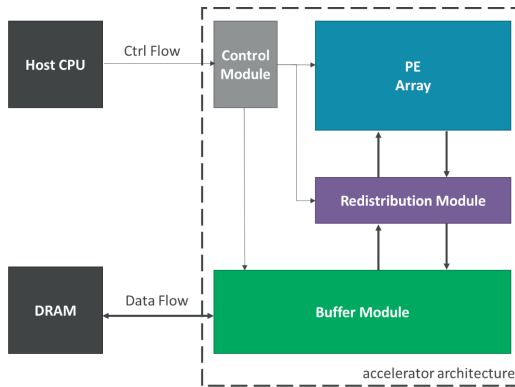


Fig. 5. Block diagram of the proposed hardware accelerator.

3.2 Processing Engine

In this paper, the hardware accelerator will use 16 PEs in the PE array for parallel computations. Two permutation blocks for each input vector deals with different data

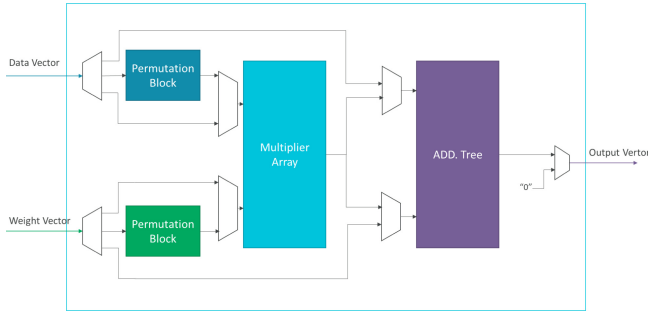


Fig. 6. Block diagram of the processing engine.

reuse schemes. There are 18 multipliers in the multiplier array responsible for two 3×3 convolutions at one cycle. Adder tree can perform different kinds of summation under different configurations. The architecture of a single PE is illustrated in Fig. 6.

Vector-Based Calculation. Each processing engine takes two vectors as input and outputs a result vector, acting like a vector processing unit. To maximize the reuse of the input vector, we carefully choose the size to be 16×16 -bit, and the permutation block helps us to rearrange the order of data in the vector. Weight vector can be an array of weight data under convolution operations or an array of pixel data when operating residual summations.

Permutation Block. Reusing data between two adjacent convolution windows is a way to improve the power efficiency of the accelerator when performing depthwise convolution, and here the permutation block is designed to undertake the job. In the “Row-first” data arrangement (please refer chapter 3.3 for more information about data arrangement in buffers), 4 pixels in the same row can be fetched in one cycle. Figure 7 shows that in 3×3 convolution with one padding, two adjacent convolution windows of calculation need 4 pixels in a row at most. Permutation blocks re-arrange data from buffers into two separate vectors for two windows and send them to processing unit. In addition, pixels in the right-most column of window are temporarily store in the module for convolution in the next cycle. The re-arrangement and temporary storage improve the data utilization of input feature map, saving power by reducing read access to buffers.

Depthwise Convolution. Depthwise convolution performs convolution for each feature map in different input channels separately. Each processing engine is in charge of all multiplications and accumulations for one input channel so 16 channels can be calculated in parallel at a time. The output vector contains two valid results from two side-by-side 3×3 windows for depthwise convolution.

Pointwise Convolution. Pointwise Convolution performs convolution operations across different input channel for input pixel data. For the case that input channels are larger than 16, the intermediate result is registered at the output stage of PE and waits for the next intermedia result in the coming round. For more efficient calculation and data fetch, 16-pixel data in data vector will be data from different feature maps but

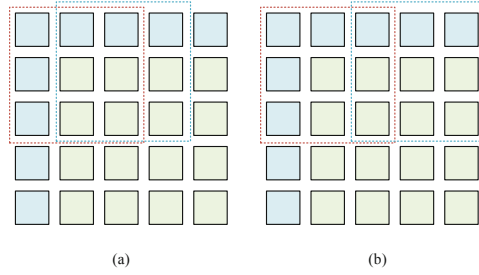


Fig. 7. Two adjacent convolution windows in different strided-convolutions. (a) is for stride-1 and (b) is for stride-2. In both figures, blue pixels stand for zero-padding pixels while green pixels are for pixels in feature maps. (Color figure online)

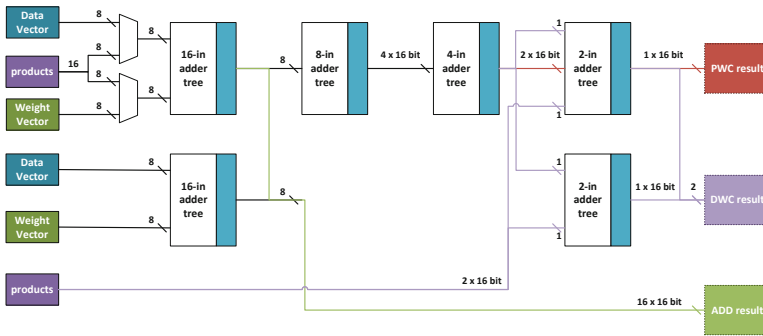


Fig. 8. Different configurations of adder tree. (Color figure online)

in the same row and column position, unlike the usual “row-first” order of data. More information about the data arrangement will be discussed in Sect. 3.3.

Residual Summation. As it is described in Sect. 2, residual summation performs at the end of every residual module. To reuse the resource of adder tree, residual enter the processing engine through weight vector and performs the summation with the result from the former layer.

Adder Tree. Adder tree is configurable to do the summation in depthwise convolution, pointwise convolution and residual addition. It is configured into various number of stages of adders for different layers, as illustrated in Fig. 8. For vector addition used in residual summation, two vector inputs coming from data vector and weight vector will enter two separate 16-in adder trees and produce a 16×16 -bit result vector, which only need 1 stage of adders. For pointwise operation, 16 products coming from the former multiplier stage will go through all the adder tree on the upper path to produce 1-element result. A little bit complicated case for depthwise convolution is that 18 products will come from the multiplier arrays (partial sums from two convolution windows), and an extra 2-in adder tree will deal with the 2 extra products (the purple lines in the figure shows how the dataflow goes in depthwise mode) in the last stage and there will be 2-element result for depthwise convolution.

ReLU. The ReLU operation, $f(x) = \max(0; x)$ is optional in the network. The process is simply replaced the negative results by zero on the output stage.

3.3 Memory Organization

The new structure in stacked hourglass network requires much dedicated design for the data flow control to efficiently make use of the memory bandwidth. Therefore, in the proposed architecture, we adapt different strategies for different situations, trying to explore the efficient data flow methods.

Table 1. Percentage of different types of operation in stacked hourglass network.

Operation type	Conv1	Conv3	Add
Percentage	90.41%	9.32%	0.27%

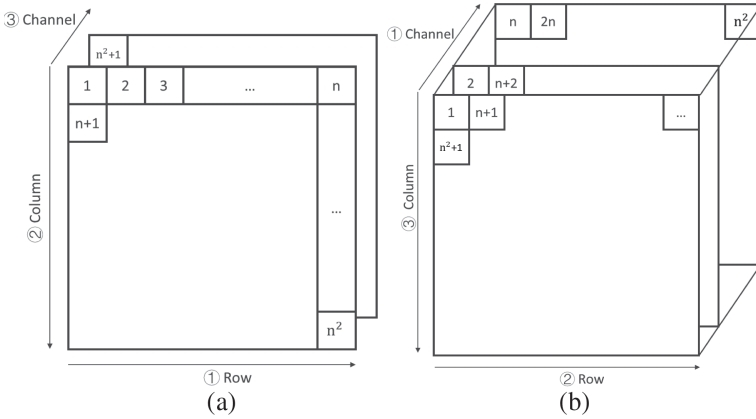


Fig. 9. “Row-first” data arrangement (a) and “Channel-first” data arrangement (b).

Data Arrangement. From a typical stacked hourglass network, we can see that the pointwise convolution occupies a large part of the whole calculations (Table 1). Pointwise convolution convolves feature maps from different channels and produces one feature map for one time. Traditional “Row-first” data arrangement (which means that data are arranged along the row direction first, then in column and channel direction successively, illustrated in Fig. 9(a)) brings trouble for this kind of convolution due to the discontinuous input data fetch, but works fine for the depthwise convolutions. “Channel-first” (which means that data are arranged along the channel direction first, then row and column direction successively, illustrated in Fig. 9(b)) arrangement is proposed to resolve the troubles. Moreover, to guarantee the computational efficiency for both depthwise and pointwise convolutions, the data arrangement is automatically exchanged during the

output stage of the processing engine. Types of upcoming layer is told to the redistribution module as well, so the results streaming out to the result buffer will change to the proper arrangement here.

In our implementation, each PE is attached with 3 buffers, an input buffer storing input activations, an output buffer collecting the output results and a weight buffer to store weights. The input buffer and the output buffer are in the same size with 4 banks each and a total of 256-bit data width while the weight buffer is with the same data width but shorter in depth. When performing depthwise convolution and standard convolution, elements in different rows can be fetched in a time enabling parallel computation for multiple MAC operations in different rows for a convolute window. In contrast, multiple elements in different channels can be fetch in a time when performing pointwise convolution in “channel-first” data arrangement and speed up the accumulation between input channels.



Fig. 10. An illustration of address mapping. Note that same color blocks in the result after upsample is represented the same origin pixel before the operation. (Color figure online)

Address Mapping for Upsample. Upsample is the operation to scale up the intermediate result to the same size of the residual so that they can perform the summation. To avoid the waste of memory space and bandwidth, address mapping method is proposed here to combine the upsample operation and the following residual summation together in the accelerator. A simple demonstration of address mapping is illustrated in Fig. 10. After upsample in the usual stride 2, one pixel is scaled up into a 2×2 result block with same value in it. When the summation tries to fetch the result of the upsample operation, it actually fetches the input of the upsample 4 times with simple address mapping. Moreover, for some specific circumstance, the mapping is further simplified to the interception and concatenation of the address without any complex address calculations.

Tiling for Data. Tiling is necessary for various sizes of input feature maps to accommodate in the on-chip buffer with fixed and limited size. Moreover, data arrangement for different layer is different as mentioned in data arrangement, direction of tiling differs as well. For operands of depthwise convolutions or residual summations, tiling is along the channel direction and for pointwise convolutions, tiling is along the row direction.

4 Network Mapper

This hardware accelerator heavily relies on the control instructions sent from the host CPU to control the behaviors and to configure the hardware accelerator. To maximize the performance for the network on the hardware, we design a dedicated network mapper for the proposed accelerator. The mapper acts as a compiler for transforming the high-level network descriptions (like excel form) to the corresponding code-like control instructions.

4.1 Overview of Network Mapper

To perform the calculation of one particular layer of stacked hourglass network, the hardware accelerator need to know the information about size of feature map, how the layer cascade, parameters for efficient tiling, etc. It is necessary to do the calculation for those parameters in advance to reduce the extra burden on the accelerator, which is the job for network mapper (Fig. 11) Network mapper first reads through the architecture of the network and do the analysis and calculations. The major job will be as follows.

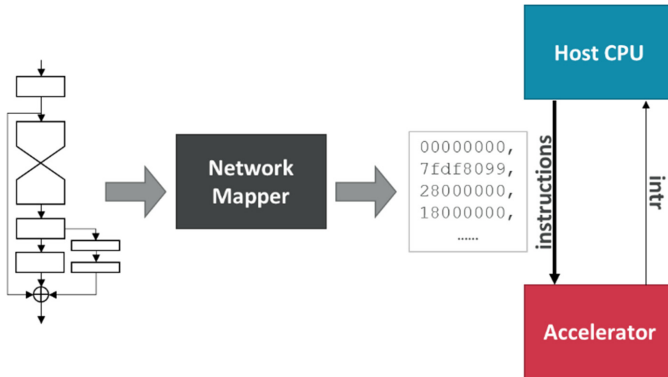


Fig. 11. The function of network mapper.

- 1) Layer Cascade: Some layers can be cascade to reduce the memory bandwidth for less intermediate results, for example, ReLU activation is performed after the calculation of the former layer, and the upsample operation is done with the following residual summation as mentioned in Sect. 3.3.

- 2) **Gather-Scatter Setup:** A buffer controller is in charge of the communication between multiple banks in the accelerator and the external memory, acting like a simplify gather-scatter DMA. Different sizes of feature map result in different transfer length for layers and they are calculated in network mapper.
- 3) **Feature Map Tiling:** Tiling is always necessary in the acceleration and the tiling method is described in Sect. 3.3. Network mapper figure out the best tiling strategy for different size of feature maps ahead and configure the accelerator through control instructions.

Once the architecture of the network is fixed, control instructions can be generated immediately. The accelerator follows the instructions to calculate for one frame and loop over the instructions for video stream. The size of the instructions for a typical stacked hourglass network is not greater than 20 KB for each frame. Due to the tiny size of the codes, they can be either stored in the on-chip memory or sent by the host CPU for convenient control scheme.

4.2 Residual Optimizing

As mentioned in Sect. 2, stacked hourglass network has more than one kind of residual summation branch for multiple resolutions. As the nested hourglass become deeper, it is impossible to store all the intermediate residual results in the on-chip memory for the limited memory space. Also, how to choose the right intermediate residual exactly for the summation has become a challenge as a residual will be used for multiple times. We proposed a method to control the residual storage inspired by the use of stacks in CPUs. When reaching a residual layer during the network structure read-in, the network mapper will record the residual level and how many times this residual result will be used, and a new memory space is allocated to store this residual. When reaching an add layer, the processing engine looks for the latest residual for sum operation, and the corresponding residual count will be decreased by one. The residual count counted down to zero means that the residual result is no longer needed, so the memory space will be set free for subsequent use. After finishing all the calculations in the network, memory space occupied by residuals should be cleaned up for all the residuals have been used and freed. The network mapper follows these principles to allocate the memory space for accelerator in advance. This method is also applied to the data flow branches of the intermediate supervision process between two stacked hourglass modules.

5 Experimental Result

The proposed accelerator architecture is implemented on the Zynq-7000 ZC706 evaluation board (XC7Z045), which contains 218,600 LUTs, 545 block RAMs and 900 DSPs. The implementation result and the performance comparison will be discussed below and then followed by a pose estimation demonstration using the proposed accelerator.

5.1 Implementation Result

We choose the number of PEs to be 16 and every PE is paired with an input buffer, a weight buffer and an output buffer. The size the input buffer and output buffer is same with 1536×256 -bit each and the size of a weight buffer is 64×256 -bit.

Table 2. Resource utilization of our implementation

Module	LUT	DSP	BRAM
PE array	22394 (10.3%)	288 (32.0%)	0 (0.0%)
Buffer	51412 (23.5%)	9 (1.0%)	448 (82.2%)
Total	73806 (33.8%)	297 (33.0%)	448 (82.2%)

Table 3. Performance comparison with other implementations

Work	[13]	Embedded CPU	Desktop CPU	GPGPU	Ours
Platform	Arm Kyro	Arm Cortex-A9	Intel i7-6800 k	Nvidia 1080 Ti	Zynq XC7Z045
Tech node (nm)	14	28	14	16	28
Clock (MHz)	2150	667	3400	1480	125
Power (W)	–	–	30	55	8.6
Problem complexity (GOP)	0.608	0.953	0.953	0.953	0.953
Computation time (ms)	75	33427	121.80	56.21	112.76
Performance (GOP/s)	8.107	0.0285	7.824	16.954	8.434

The resource utilization is shown in Table 2. The stacked hourglass algorithm needs 0.953 GOPs, mostly of them are multiplications and summations. Our system achieves an average performance of 8.434 GOP/s and the frame rate reach 8.85 fps.

Since our design is for the embedded applications, and also is the first implementation for the hourglass network as far as we know, the performance comparison will be made with the embedded CPU within the Zynq XC7Z045, which is an ARM Cortex-A9 core. We also make the comparison with the performance mentioned in [15] for the similar algorithm structure of depthwise separable convolution. Our implementation achieves the average performance of 8.434 GOP/s, which is $296\times$ faster than the embedded CPU.

Regardless of extra memory access for the multiple resolution summation of residual blocks in stacked hourglass network, our implementation can still slightly override the performance to MobileNetV2 implemented in Kryo CPU. In addition, we also run the same hourglass network on the modern desktop CPU and GPGPU. It's surprise that our implementation does better jobs than the desktop CPU, by $1.08\times$ and $3.76\times$ on calculation speed and power efficiency respectively. Our works also wins $3.18\times$ power efficiency when comparing to GPGPU. The performance of our system and different devices is shown in Table 3.

Considering the nature of the stacked hourglass network that residual results with multiple resolution are created and accumulated during the inference phase, frequent off-chip memory access seems inevitable and it's the main reason for the low utilization of PE in our work. We consider it unfair to compare our works with other works implemented on the same ZC706 which runs the holistic network structures without branches.

5.2 Application

We also use the proposed hardware accelerator to build an embedded system for real-time pose estimation demo on ZC706 evaluation board as well. The system is in the heterogeneous architecture. The whole stacked hourglass network is calculated within the proposed hardware accelerator while the embedded ARM Cortex-A9 dual core CPU is responsible for the control of the video capture, weights preload, image pre-processing and the HDMI display. The accelerator runs at the frequency of 125 MHz. The block diagram of the whole system is presented on Fig. 12.

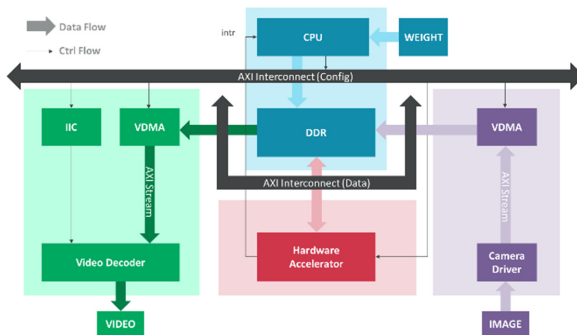


Fig. 12. Block diagram of the demo system.

Figure 13 shows the overview of the system and the pose estimation application demo. As shown on the monitor, the system is able to reproduce the skeleton and the key points of the human body with relatively high precision.

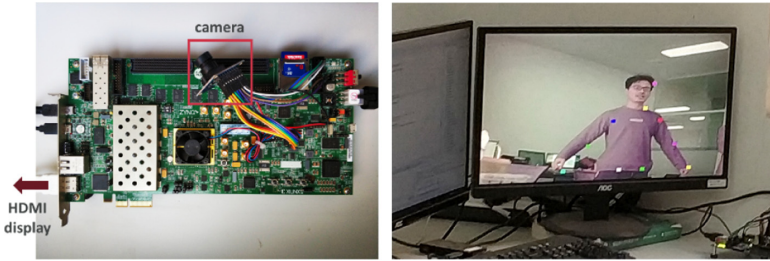


Fig. 13. FPGA evaluation board used for the pose estimation demo (left) and the demonstration (right).

6 Conclusion

In this article, a hardware accelerator implementation is purpose for the stacked hourglass network. The architecture of the accelerator is optimized for the depth separable convolutions and the multiple resolution residual summations that are frequently seen in the stacked hourglass network. With such dedicated design, high accuracy pose estimation applications can be done on the portable device. As an example, our accelerator implemented on Zynq XC7Z045 can achieve an average performance of 8.434 GOP/s with high power efficiency of 0.981 GOP/s/W under the 125 MHz working frequency. Also, a pose estimation demo is also presented here using the purpose hardware design.

References

1. Newell, A., Yang, K., Deng, J.: Stacked Hourglass Networks for human pose estimation. [arXiv:1603.06937v2](https://arxiv.org/abs/1603.06937v2) [cs. CV], July 2016
2. Chen, Y., Krishna, T., Emer, J.S., Sze, V.: Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circ.* **52**(1), 127–138 (2017)
3. Luo, T., et al.: DaDianNao: a neural network supercomputer. *IEEE Trans. Comput.* **66**(1), 73–88 (2017)
4. Qiu, J., et al.: Going deeper with embedded FPGA platform for convolutional neural network. In: *Proceeding of FPGA*, Monterey, CA, USA, pp. 26–35 (2016)
5. Gokhale, V., Zaidy, A., Chang, A.X.M., Culurciello, E.: Snowflake: an efficient hardware accelerator for convolutional neural networks. In: *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS 2017)*, pp. 1–4 (2017)
6. Meloni, P., et al.: NEURAghe: exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on Zynq SoCs. *ACM Trans. Reconfigurable Technol. Syst.* **11**(3), 1–24 (2018)
7. Su, J., et al.: Neural network based reinforcement learning acceleration on fpga platforms. *ACM SIGARCH Comput. Archit. News* **44**(4), 68–73 (2016)
8. Guo, K., et al.: Angel-eye: a complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **37**(1), 35–47 (2018)
9. Kim, D., Ahn, J., Yoo, S.: ZeNA: zero-aware neural network accelerator. *IEEE Des. Test* **35**(1), 39–46 (2018)
10. Aimar, A., et al.: NullHop: a flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE Trans. Neural Netw. Learn. Syst.* **30**(3), 644–656 (2019)

11. Bianco, S., Cadene, R., Celona, L., Napolitano, P.: Benchmark analysis of representative deep neural network architectures. *IEEE Access* **6**, 64270–64277 (2018)
12. Su, J.: Artificial neural networks acceleration on field-programmable gate arrays considering model redundancy. Imperial College London Ph.D. thesis (2018)
13. Lin, X., Yin, S., Tu, F., Liu, L., Li, X., Wei, S.: LCP: a layer clusters paralleling mapping method for accelerating inception and residual networks on FPGA. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, pp. 1–6 (2018)
14. Ma, Y., Kim, M., Cao, Y., Vrudhula, S., Seo, J.: End-to-end scalable FPGA accelerator for deep residual networks. In: 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, pp. 1–4 (2017)
15. Howard, A.G., et al.: MobileNets: efficient convolutional neural networks for mobile vision applications. [arXiv:1704.04861](https://arxiv.org/abs/1704.04861) [cs], April 2017
16. Chollet, F.: Xception: deep learning with depthwise separable convolutions. [arXiv:1610.02357v3](https://arxiv.org/abs/1610.02357v3) [cs], April 2017
17. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. [arXiv:1512.03385v1](https://arxiv.org/abs/1512.03385v1) [cs. CV], December 2015
18. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.: MobileNetV2: inverted residuals and linear bottlenecks. [arXiv:1801.04381](https://arxiv.org/abs/1801.04381) [cs. CV], January 2018
19. Venkataramani, S., et al.: ScaleDeep: a scalable compute architecture for learning and evaluating deep networks. In: ISCA 2017 Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 13–26 (2017)



PRBN: A Pipelined Implementation of RBN for CNN Training

Zhijie Yang¹, Lei Wang¹(✉), Xiangyu Zhang¹, Dong Ding¹, Chuan Xie², and Li Luo¹

¹ College of Computer Science and Technology, National University of Defense Technology, Changsha, China

Leiwang@nudt.edu.cn

² College of Computer Science and Technology, Southwest Minzu University, Chengdu, China

Abstract. Recently, training CNNs (Convolutional Neural Networks) on-chip has attracted much attention. With the development of the CNNs, the proportion of the BN (Batch Normalization) layer's execution time is increasing and even exceeds the convolutional layer. The BN layer can accelerate the convergence of training. However, little work focus on the efficient hardware implementation of BN layer computation in training. In this work, we propose an accelerator, PRBN, which supports the BN and convolution computation in training. In our design, a systolic array is used for accelerating the convolution and matrix multiplication in training, and RBN (Range Batch Normalization) array based on hardware-friendly RBN algorithm is implemented for computation of BN layers. We implement PRBN on FPGA PYNQ-Z1. The working frequency of it is 50 MHz and the power of it is 0.346 W. The experimental results show that when compared with CPU i5-7500, PRBN can achieve $3.3\times$ speedup in performance and $8.9\times$ improvement in energy efficiency.

Keywords: Deep convolutional neural network · Training · Batch normalization · Accelerator

1 Introduction

CNNs achieve high accuracy in image recognition tasks [1]. Training CNNs is more difficult than their inference. The reason is that training includes forward propagation, error backpropagation, and weight update, whereas inference only includes forward propagation. The processes of error backpropagation and weight update is time consuming. Moreover, the convergence speed of training is slow, because most of the training uses SGD (Stochastic Gradient Descent) solver [2] which leads to the gradient explosion and dispersion problem.

BN [3] layer is one of the most important layer in CNNs. It can prevent gradient explosion or dispersion and accelerate the convergence speed of CNN training. BN layer is getting more important than ever before. In conventional CNNs, the execution time of convolution layers accounts for more than 90% of the total execution time. But in the state of the art CNN such as DenseNet-121 [4], the execution time of non-convolutional layers (primarily BN layers) even

exceeds the time of convolutional layers in training [5]. As shown in Fig. 1, we also observe that the execution time of BN layers exceeds convolutional layers in some layers in forward propagation of training MobileNet [6]. The reason is that BN layer’s computation includes complex computations of statistics such as the mean and variance of cumulative sums in each mini-batch which will be introduced in detail in Sect. 2. This process includes a large number of multiplication, square root and addition operations as well as complex control flow. However, little work focus on the efficient hardware implementation of BN layer computation in training.

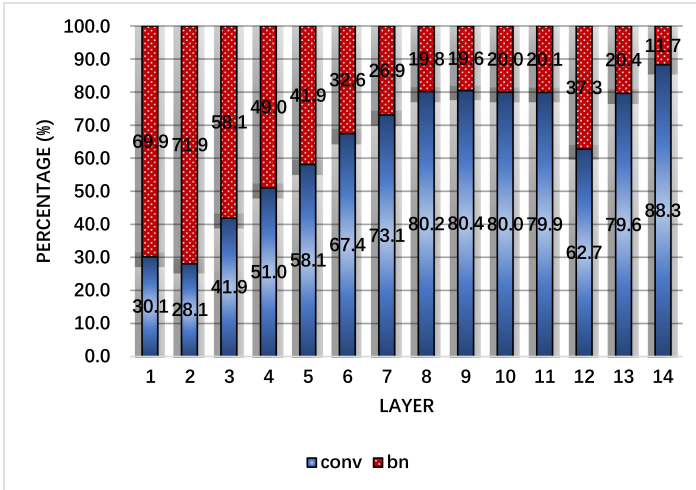


Fig. 1. Execution time breakdown of BN layer and convolutional layer in MobileNet V1.

Thus, in this paper, we focus on accelerating the BN layer and convolution computations that dominate the training workload. Our main contributions are as follows:

- Based on the hardware-friendly RBN algorithm [7], we propose a pipelined hardware implementation for forward and backforward propagation of BN layer computation.
- We integrate the above module and a systolic array [8] for accelerating convolution and matrix multiplication in forward propagation and backpropagation in training into an accelerator, PRBN, to accelerate CNN training. By this way, our accelerator can support the workload that takes up most of the training execution time, i.e. BN and convolution computation.

We implement the accelerator in RTL-level code and deploy it on FPGA PYNQ-Z1. The experimental results show that when compared with CPU i5 7500, PRBN can achieve $3.3\times$ speedup in performance and $8.9\times$ improvement in energy efficiency.

2 Background

The algorithm used in the training of CNN is the backpropagation algorithm [9]. It consists of three processes, forward propagation, backpropagation and error update. In this section, we introduce the computation of the convolutional layer and BN layer which dominate the training execution time.

2.1 CNN Training

Forward Propagation of Convolutional Layer and Fully-Connected Layer. For the convolutional layer and the fully connected layer, the input multiplies weights to get partial sums and partial sums are accumulated to get activation. The computation process of activation is as follows:

$$z^l = W^l * a^{l-1} + b^l \quad (1)$$

$$a^l = f(z^l) \quad (2)$$

In the above equation, l is the layer index, z is the sum, W is the weight matrix, a is the activation, b is the layer bias parameter, and f is the activation function. The main computation of the convolutional layer and the fully connected layer in the forward propagation is convolution and matrix multiplication.

Forward Propagation of BN Layer. Since the training dataset is divided into several mini-batches, the function of the BN layer is to form the mean and standard deviation of each mini-batch and then normalize all of the data in the mini-batch according to the obtained mean and standard deviation. The BN layer algorithm in training is shown in Algorithm 1.

Algorithm 1. Algorithm of batch normalization [3].

Input: Values of z over mini-batch n : $z_1 \dots z_m$;

Parameters to be learned: γ, β

Output: $\{y_i = BN_{\gamma, \beta}(z_i)\}$

$$\mu = \frac{1}{m} \sum_{i=1}^m z_i \quad // \text{mini-batch mean}$$

$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (z_i - \mu)^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(z_i) \quad // \text{scale and shift}$$

In Algorithm 1, the inputs are sums of a mini-batch produced by an output neuron in the previous layer. The output of the batch normalization is y . μ is the mean of the mini-batch, and m is the size of the mini-batch. σ is the variance of this mini-batch. x is the result of normalizing the input to $N(0, 1)$. ϵ is a

small constant that prevents the denominator from being 0. y is the result of adjusting for x . γ and β are parameters to be learned according to the chain rule by backpropagation after the forward propagation of one mini-batch is done.

In addition to the mean and variance of each mini-batch, the BN layer also needs to form the global mean and global variance of all mini-batches. They are formed as follows:

$$\mu_{global}^n = \mu^n * \frac{1}{N} + \mu_{global}^{n-1} * \frac{N-1}{N} \quad (3)$$

$$(\sigma^2)_{global}^n = (\sigma^2)^n * \frac{1}{N} + (\sigma^2)_{global}^{n-1} * \frac{N-1}{N} \quad (4)$$

In the above formula, μ_{global}^n is the partial result of the global mean updated in mini-batch n . N is the number of mini-batches. σ is the standard deviation, and its square is the variance.

Back Propagation. Once the forward propagation of a mini-batch is completed, the difference between the results and the labels can be obtained based on the cost function. And the error map can be obtained as well. The error map of the output layer is formed as follows:

$$\delta^L = \nabla C \odot f'(z^L) \quad (5)$$

In the above formula, δ is the error map, C is the cost function, f is the activation function, z is the sum before activation, and L is the output layer. f' is the derivative of f . The error maps are propagated forward layer by layer. The error maps of the previous layer are formed as follows:

$$\delta^{l-1} = ((W^l)^T * \delta^l) \odot f'(z^{l-1}) \quad (6)$$

In the above formula, l is the layer index, W is the weight matrix, and T is the transpose sign. Matrix multiplication or convolution is performed between the weight matrix and the error map. The update values of the weights and the bias parameters are formed as follows:

$$\Delta w_{ij} = \frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1} * \delta_i^l \quad (7)$$

$$\Delta b_i = \frac{\partial C}{\partial b_i^l} = \delta_i^l \quad (8)$$

In the above formula, w is the weight, a is the activation, b is the bias parameter, and i and j are the indexes of the weight matrix.

2.2 Range Batch Normalization

Banner et al. [7] first proposed the RBN algorithm. The main idea is that the standard deviation σ used in conventional BN algorithm can be approximated by the

product of correlation coefficient $c(m)$ and the range of input values. It is based on the theory that, if the input data are assumed to follow Gaussian distribution, the range of the input is highly correlated with the standard deviation σ .

$$\hat{x}_i = \frac{z_i - \mu}{c(m) \cdot \text{range}(z_i - \mu)} \quad (9)$$

RBN in Training. The only difference between RBN and BN in forward propagation of training is the normalization. The normalization of RBN in training is calculated as Eq. 9. As shown in Eq. 9, $c(m) = 2\sqrt{\ln(m)}$, where m is the mini-batch size. And $\text{range}(z) = \max(z) - \min(z)$. With RBN, the computation of variance is eliminated, which includes a lot of multiplication and square root operations. Besides, RBN solves the problem of data overflowing in variance computation, so that lower data width can be applied to training. Therefore, RBN saves hardware resources and reduces the execution time of the BN layer in training. But RBN will not reduce the network accuracy [7].

RBN in Inference. The process of RBN in inference is similar to forward propagation of RBN in training, i.e. Algorithm 1. The difference is that the mean μ and standard deviation σ used in RBN in inference are calculated from all training pictures after training, which are called the global mean and the global standard deviation respectively. So we merged and simplified the coefficients as a and b calculated as follows:

$$a = \frac{\gamma}{\sigma_{global}} \quad (10)$$

$$b = \beta - a * \mu_{global} \quad (11)$$

3 Proposed Architecture

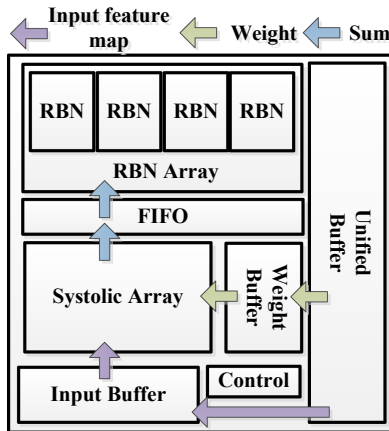


Fig. 2. Block diagram of the overall architecture.

3.1 Overview

In this paper, we integrate systolic array and RBN array into an accelerator design which can support the BN and convolution operations in training. Due to the high parallelism and high data reusability, systolic array is suitable for serving as the core computing unit of CNNs to support convolution computation, which is computationally intensive. Besides, because the BN function is generally performed after the convolutional function, the inputs of BN function are sums produced systolic array. Therefore, using systolic array naturally matches the RBN array which receives results generated from systolic array.

Figure 2 is the overall architecture of our implementation. It has three components. A systolic array with input and weight buffers for the convolution computation, an RBN array for the BN computation, and the unified buffer. The systolic array is used to compute convolution and matrix multiplication. RBN array is used to support the BN layer computation. The unified buffer is used for data exchanging with off-chip host. A FIFO serves as an intermediate buffer between systolic array and RBN array so that these two parts can be executed in the form of the pipeline.

Our accelerator has four working modes for different processes in training and inference. The mode selection signal is generated by the controller according to the current working process.

Mode 1 is for the forward propagation of training. In this mode, the activation and weights are fed into the systolic array to compute the sums. Sums enter RBN array to be normalized. Mode 2 is for the backpropagation of training. In this mode, errors and weights are fed into the systolic array to compute the errors of the previous layer. RBN array performs updates to the γ and β . Mode 3 is for the inference. In this mode, the input feature maps and weights are input to the systolic array to compute the sums. Sums enter RBN array to be normalized according to the parameters obtained in training. Mode 4 is for the condition which the systolic array works but the RBN array does not. This mode is corresponding to the network structure without the BN layer.

We will introduce the design of the systolic array and the RBN array in details in the following sections.

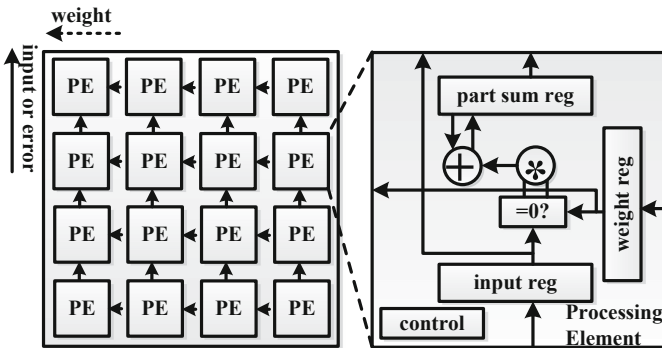


Fig. 3. Block diagram of the systolic array and details of PE design.

3.2 Systolic Array

In our design, an 8×8 systolic array is used to accelerate the convolution and matrix multiplication in forward propagation and backpropagation in training.

The systolic array receive weights, input feature maps or errors from its surrounding buffer. In the array, each processing elements (PE) is connected with its left, upper, lower PEs to deliver weights, input feature maps and accumulation sums respectively. And each PE has the function of multiplication and accumulation (MAC).

The workflow of the systolic array is as follows. As shown in Fig. 3, in the forward propagation (mode 1, 3 and 4), the activations are transmitted into the array from the buffer located below the array, passed vertically and reused between PEs. Weights are transmitted into the array from the buffer on the right, passed horizontally, from right to left, and reused between PEs. The obtained sums are transmitted vertically and finally transferred out of the array.

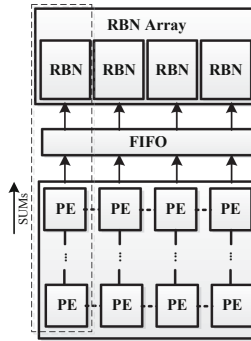


Fig. 4. The shared relationship between PEs and RBN units and the dataflow of the sums.

During the backpropagation (mode 2), errors are transmitted from the input buffer into the systolic array. It is passed in the same way as activation. The weight is passed in the same way as it does in forward propagation.

The function of the PE is to receive input from the adjacent units below and to the right. And then cache the input and pass it to the adjacent units above and the left in the next cycle. The two inputs are multiplied and added together with the partial sum in PE. Before multiplication, PE will determine if the input is 0 to skip unnecessary operations to save energy. Finally, when the control signal is received, PE outputs the sum.

It is worth noting that in our previous work [10], we integrated the RBN unit directly into the PE. However, it brought too much overhead. In addition, the sums are transmitted in columns from the systolic array. So in our architecture, for the systolic array, the PEs in each column share a RBN unit. As shown in Fig. 4, sums produced by PEs in the same column are transferred to the same RBN unit to carry BN computation. And in order to reduce the exchange of

data with off-chip host and increase the reuse of on-chip data, our arrangement strategy is based on [11].

3.3 Implementation of Batch Normalization

In this design, RBN array is implemented for the computation of batch normalization layer. It has two modules to perform batch normalization computation in inference and training respectively. And there is a multiplexer which can output results according to the mode control signal.

RBN in Training. The computation of the BN layer in the forward propagation of training (mode 1) can be divided into three parts: forming statistics, normalization, and updating partial results of global statistics. They are executed by three pipelines respectively.

The workflow and the coordination of these three pipelines is described as follows.

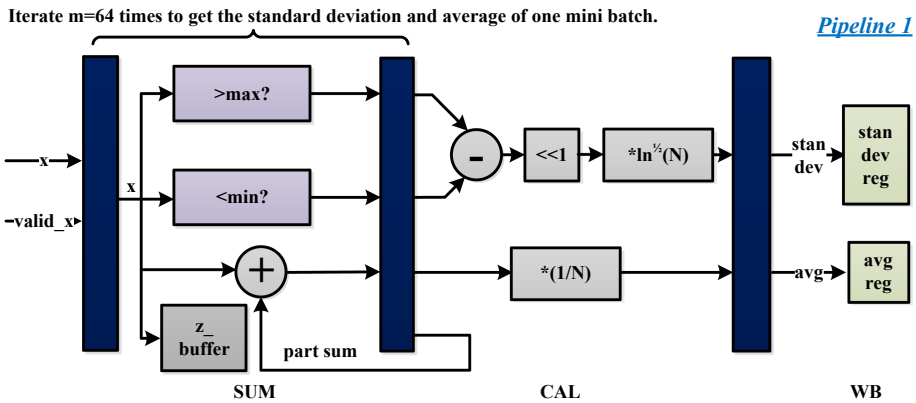


Fig. 5. Block diagram of the pipeline 1.

The pipeline 1 is used to compute the statistics, i.e. the mean and standard deviation. As shown in Fig. 5, the first stage receives the sums and their valid signals from the systolic array, then updates the maximum and minimum values, accumulates, and stores inputs in the buffer in stage SUM. When all the data for the entire mini-batch has been entered, the maximum, minimum and sum of input can be obtained. So the mean and the standard deviation can be computed based on the mentioned equations of the RBN algorithm. At last, they are stored in special registers waiting to be used in the next two pipelines.

The pipeline 2 is used for normalization. As shown in Fig. 6, after the execution of the pipeline 1, the pipeline 2 reads the mini batch’s statistics from the registers and the stored sums from the buffer. First, the sums are adjusted to

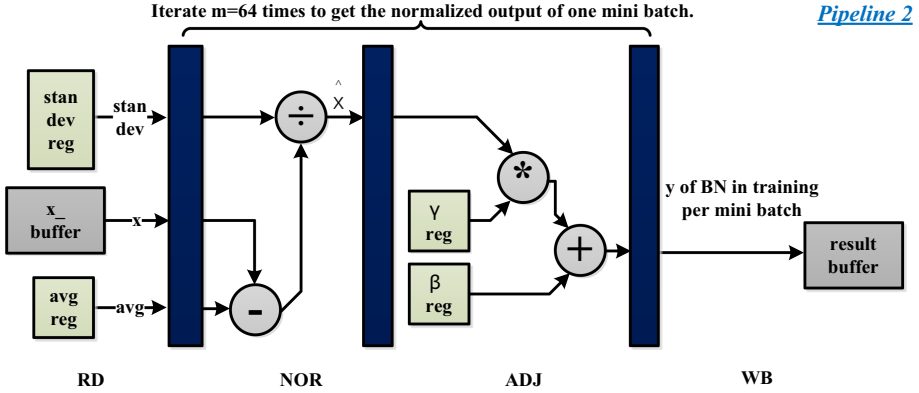


Fig. 6. Block diagram of the pipeline 2.

$N(0, 1)$, i.e. the normal distribution with a mean of 0 and a variance of 1. And then they are adjusted by γ and β , which allows them to be distributed in a linear region of the activation function, increasing the effectiveness of activation. Finally, the normalized results are either stored in the result buffer or used as input of the next layer.

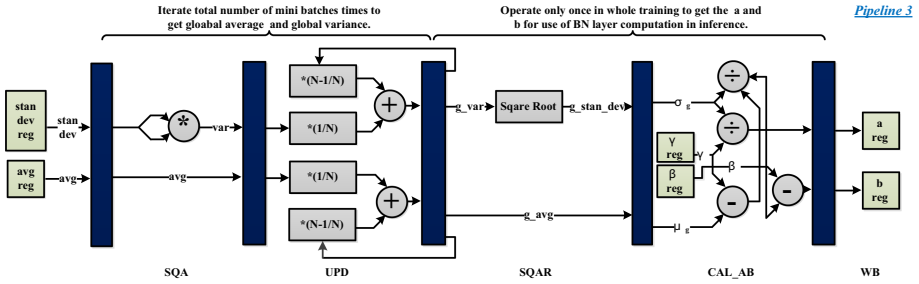


Fig. 7. Block diagram of the pipeline 3.

The pipeline 3 is used to update partial results of the global statistics and finally get the global statistics. As shown in Fig. 7, after the execution of the pipeline 1, pipeline 3 reads the mini batch’s statistics from the registers. Based on Eq. (3) and Eq. (4), the partial results of the global mean and global variance are updated in stage UPD and saved. After all the mini-batches data have been entered, the global mean and global variance can be obtained. According to Eq. (10) and Eq. (11), the simplified parameters of the BN layer in inference, a and b can be computed and saved in stage CAL_AB (Fig. 8).

The BN layer is simple during the backpropagation of training (mode 2), which only needs to update the parameters γ and β stored in their registers respectively according to the chain rule.

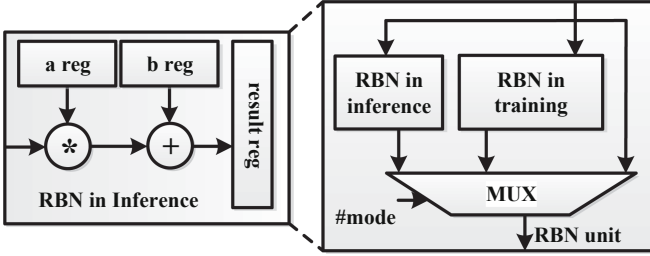


Fig. 8. Block diagram of the RBN unit.

RBN in Inference. After the parameters a and b are obtained in training, the BN layer in inference (mode 3) performs multiplication and addition according to the last step of the BN algorithm, that is, scale and shift. Thus this module has two registers to store a and b .

4 Evaluation

4.1 Experiment Setup

In this work, the PRBN is implemented in the RTL-level Verilog code. The detailed hardware configuration is as follows. The weight buffer and input buffer is 1.1 KB and the z buffer in each RBN unit is 128 B. The unified buffer is 18 KB. The size of the systolic array is $8 * 8$. The size of the RBN array is $1 * 8$. The operands are 16-bit, fixed-point. The maximum size of a mini-batch supported by a RBN unit is 64. The simulation and synthesis tool is Vivado 2017.02. The FPGA platform was PYNQ-Z1. The network we use is MobileNet V1 [6]. The software framework used for CPU baseline is PyTorch in CPU version. The dataset we use is CIFAR-10.

In the experiment, we show a comparison of the hardware resource overhead of different parts of the PRBN for reference to the discussion of extensibility. Then we discuss the comparison of the performance of RBN array and CPU in BN layer computation in training. At last, we discuss the performance and energy efficiency of the PRBN compared to CPU during training.

Table 1. Hardware utilization comparison of implementations.

Module	<i>LUT</i>	<i>FF</i>	<i>DSP</i>	<i>LUTRAM</i>
Systolic array	1643	3775	64	0
RBN array	6496	2576	32	88
Unified buffer	26720	34055	0	0
FIFO	776	599	0	0
Total	35635	41005	96	88
Utilization (%)	67.0	38.5	43.6	0.5

4.2 Experiment Results

Resource Utilization. For the researcher to understand the hardware implementation overhead of each module of the whole accelerator, we show the resource utilization of each module in Table 1. PYNQ-Z1 is a low-cost device in the ZYNQ family and has ultra-low power, making it suitable for edge devices.

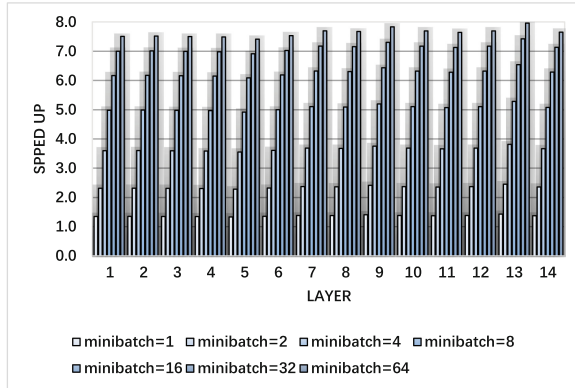


Fig. 9. The speedup of the RBN array to the CPU in the computation of the BN layer in forward propagation of the training.

Performance and Energy Efficiency. To demonstrate the speedup of our RBN implementation to the CPU I5 7500, we compare their performance at each layer of the network. As shown in Fig. 9, the speedups are similar at all

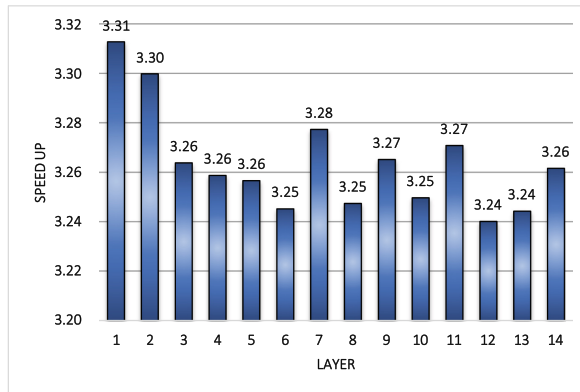


Fig. 10. The speedup of PRBN to the CPU in training of all layers of MobileNet V1, where minibatch = 8.

layers. Under different minibatch conditions, the speed up keeps rising. But the rising trend gradually slows down. Because our implementation is pipelined, the larger the size of the minibatch, the more benefits our implementation will get. And this phenomenon is determined by the characteristics of the pipeline.

Figure 10 shows the performance of the PRBN versus the CPU. Our implementation can achieve $3.24\times\text{--}3.31\times$ speedup over i5 7500 CPU@3.40GHz. Our implementation’s working frequency is 50 MHz, its power is 0.346 W and the energy efficiency is 28.9 Gops/W. Thus, it can achieve $8.5\times$ energy efficiency to CPU i5 7500.

Figure 11 shows the energy breakdown of each module of our engine at each layer of the network. We choose mini-batch as 8 because if the mini-batch size is too big, there will be much overhead of data exchanging for that storage resource is limited on-chip. It can be found in Fig. 9 that the energy of the systolic array domains. Because systolic array has 64 processing elements for multiplication and accumulation, its power consumption and energy consumption are relatively high. The RBN array has a lower energy footprint because it is pipelined and uses the RBN algorithm, which reduces a lot of multiplication and square root operations compared with the conventional BN algorithm.

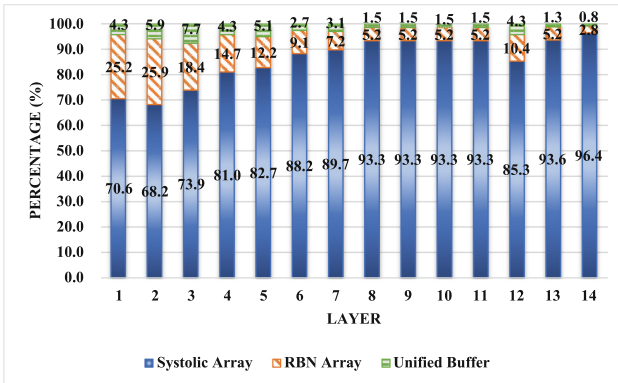


Fig. 11. The energy breakdown of the PRBN in training all layers on MobileNet V1, where minibatch = 8.

5 Related Works

There are lots of previous work aiming at deep learning efficient hardware implementation on edge devices. However, to the best of our knowledge, researchers have not paid enough attention to efficient hardware implementation of BN layer computation. Whatmough, P, N. et al. [12] implement efficient hardware for mobile computer vision via transfer learning, but it only supports the inference (Table 2).

Sledevic, Tomyslav, et al. [14] integrates multiplication and accumulation (MAC) and computation of the BN layer into a DSP and verifies the effect on

Table 2. Comparison of related work and our design.

Work	<i>Freq.</i>	<i>Prec.</i>	<i>Power</i>	Platform	BN(Inf)	BN(Tra)
[13]	200M	8 bit	12.4 W	VU9P	No	No
[14]	224M	16 bit	N.A.	ARTIX7	Yes	No
Ours	50M	16 bit	0.35 W	PYNQ-Z1	Yes	Yes

FPGA. But it only supports BN computation in inference. Xiong, Feng, et al. [13] focus on the training of efficient compact networks and design efficient hardware for it and Xie, F. et al. [15] use the edge server and FPGA for collaborative training to improve training efficiency and reduce training time. However, none of them focus on the efficient hardware implementation of BN layer computation in training.

6 Discussion

Our implementation provides a possible solution for low-power and resource-constrained edge devices such as smart cameras to train CNNs. And recently training CNNs on edge devices attracts researchers’ attention [13, 15]. There are three reasons for this. The first reason is the domain shift problem [16]. Domain shift problem refers to the severe reduction in the recognition accuracy of CNNs deployed on edge devices due to environmental or equipment problems. So re-training is necessary. An example of the domain shift problem is the viewpoint problem [17]. It refers to the recognition failure caused by the difference between the shooting angles of the pictures in the training dataset and the shooting angles in the actual inference. Therefore, retraining the network with the multi-angle datasets collected by the edge devices can improve the recognition accuracy. Because these new datasets are similar to the original training dataset, so a lot of low-level information can be shared. The second reason is user privacy [18]. Sensitive information, such as users’ faces, cannot be transmitted over the network and must remain local. At this point, training on edge avoids data leakage so it protects users’ privacy. The third reason is the transfer delay. Because of the rise of the 5G and the IoTs, more data are created by widely distributed edge devices instead of large-scale cloud data centers [19]. If all of the training tasks are performed on cloud servers, a significant transfer delay will be introduced, thereby reducing the quality of service.

7 Conclusion

CNNs achieve high accuracy in image recognition tasks. Because of transmission latency, privacy, and domain shift problems, researchers begin to focus on training CNNs on edge devices. However, training is more complex and requires more computation and storage resources than inference. Although the BN layer

can accelerate the convergence of training, it contains a large number of computations and complex control processes and conventional edge devices still use inefficient CPUs for BN layer computation. In this work, we propose PRBN, which can support the BN and convolution in training for edge devices. The systolic array is used for accelerating the convolution and matrix multiplication in forward propagation and backpropagation in training. And pipelined RBN array based on the hardware-friendly RBN algorithm is used to support the computation of BN layers. The experimental results show this chip achieves $3.3\times$ speedup in performance and $8.9\times$ improvement in energy efficiency when compared with CPU i5 7500. In our future work, we will focus on implementing an end-to-end coprocessor which supports training on edge devices.

References

1. He, K., et al.: Deep residual learning for image recognition. In: Computer Vision and Pattern Recognition, pp. 770–778 (2016)
2. Bottou, L.: Stochastic gradient descent tricks. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) *Neural Networks: Tricks of the Trade*. LNCS, vol. 7700, pp. 421–436. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35289-8_25
3. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning, pp. 448–456 (2015)
4. Huang, G., et al.: Densely connected convolutional networks. In: Computer Vision and Pattern Recognition, pp. 2261–2269 (2017)
5. Jung, W., et al.: Restructuring batch normalization to accelerate CNN training. arXiv preprint [arXiv:1807.01702](https://arxiv.org/abs/1807.01702) (2018)
6. Howard, A.G., et al.: Mobilenets: efficient convolutional neural networks for mobile vision applications. arXiv preprint [arXiv:1704.04861](https://arxiv.org/abs/1704.04861) (2017)
7. Banner, R., et al.: Scalable methods for 8-bit training of neural networks. In: Neural Information Processing Systems, pp. 5145–5153 (2018)
8. Kung, H.T., Leiserson, C.E.: Systolic arrays (for VLSI). In: *Sparse Matrix Proceedings*, vol. 1, pp. 256–282. Society for Industrial and Applied Mathematics, Philadelphia (1978)
9. LeCun, Y., et al.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
10. Yang, Z., et al.: Bactran: a hardware batch normalization implementation for CNN training engine. *IEEE Embed. Syst. Lett.* 1 (2020)
11. Yang, Z., et al.: Systolic array based accelerator and algorithm mapping for deep learning algorithms. In: Zhang, F., Zhai, J., Snir, M., Jin, H., Kasahara, H., Valero, M. (eds.) *NPC 2018*. LNCS, vol. 11276, pp. 153–158. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05677-3_16
12. Whatmough, P.N., et al.: FixyNN: efficient hardware for mobile computer vision via transfer learning. arXiv preprint [arXiv:1902.11128](https://arxiv.org/abs/1902.11128) (2019)
13. Xiong, F., et al.: Towards efficient compact network training on edge-devices. *IEEE Computer Society Annual Symposium on VLSI*, pp. 61–67 (2019)
14. Sledevic, T.: Adaptation of convolution and batch normalization layer for CNN implementation on FPGA. In: 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream), pp. 1–4. IEEE (2019)

15. Xie, F., et al.: Edge intelligence based co-training of CNN. In: 2019 14th International Conference on Computer Science & Education (ICCSE), pp. 830–834. IEEE (2019)
16. Donahue, J., et al.: DeCAF: a deep convolutional activation feature for generic visual recognition, vol. 50, no. 1, pp. 1–647 (2013)
17. Kukreja, N., et al.: Training on the edge: the why and the how. In: International Parallel and Distributed Processing Symposium, pp. 899–903 (2019)
18. Paul, G., Irvine, J.: Privacy implications of wearable health devices. In: Proceedings of the 7th International Conference on Security of Information and Networks. ACM (2014)
19. Deng, S., et al.: Edge intelligence: the confluence of edge computing and artificial intelligence. *Networking and Internet Architecture*. arXiv (2019)

Processor, Memory, and Storage Systems Architecture



Network-on-Chip Aware Task Mappings

Xiaole Sun¹, Yong Dong¹, Juan Chen^{1(✉)}, and Zheng Wang²

¹ National University of Defense Technology, Changsha, China
{sunxiaole18,yongdong,juanchen}@nudt.edu.cn

² School of Computing, University of Leeds, Leeds, UK
z.wang5@leeds.ac.uk

Abstract. Energy and power density have forced the industry to introduce many-cores where a large number of processor cores are integrated into a single chip. In such settings, the communication latency of the network on chip (NoC) could be performance bottleneck of a multi-core and many-core processor. Unfortunately, existing approaches for mapping the running tasks to the underlying hardware resources often ignore the impact of the NoC, leading to sub-optimal performance and energy efficiency. This paper presents a novel approach to allocating NoC resource among running tasks. Our approach is based on the topology partitioning of the shared routers of the NoC. We evaluate our approach by comparing it against two state-of-the-art methods using simulation. Experimental results show that our approach reduces the NoC communication latency by 5.19% and 2.99%, and the energy consumption by 17.94% and 12.68% over two competitive approaches.

Keywords: Network on Chip · Performance optimization · Many-cores

1 Introduction

The network-on-chip (NoC) is an essential component of multi-core processor architectures. As parallelism is the best way to utilize multi-cores, parallel workloads are now commonplace on such systems. In such a setting, the NoC is often a performance bottleneck of a multi-core system and responsible for performance slowdown of parallel workloads [1]. The NoC is also a major energy consumer of modern multi-core systems. It can consume over 28% of the total energy consumption of a multi-core processor [2], and even account for over 40% of the CPU energy consumption for multimedia applications [3]. As we are moving into a many-core era, with an increasing number of processor cores integrated into a single chip, the NoC will play an increasingly important role for performance and energy optimization of computing systems.

There have been efforts on exploring hardware and software techniques to perform performance and energy optimization, specifically targeting the NoC. For example, Chen *et al.* [4] reduce the power consumption of the NoC, by closing idle routers to without blocking communication. Other works exploit a software-centric technique to partition the router resources of the NoC among running tasks [5]. Software-based approaches have the advantage of not requiring hardware modification and can work on commercial off-the-shelf chips.

Existing work on task mapping often ignores the real-time occupation of routers of an NoC. This is a significant drawback for multi-programmed workloads, where multiple tasks or jobs use the *shared routers* concurrently. In such scenarios, existing approaches can over-subscribe the shared resources, leading to resource contention and overall performance slowdown and increased energy consumption for competing workloads.

Because of the subtle interaction among concurrently running tasks, it is important to consider the occupancy of shared routers for resource allocation. The key to minimize network congestion of the NoC is to reduce the overlap in using shared routers among concurrently running tasks. Doing so can reduce communication latency and the related energy consumption of the NoC.

This paper presents a novel software-based approach to perform power and performance optimization for the NoC. Our work dynamically allocates computing resources to match the concurrent tasks to the underlying hardware to minimize the share of routers among running tasks. We achieve this by exploiting the NoC topology to perform shared router resource partition. By always trying to assign idle routers first, our approach can reduce the resource contention, which in turn leads to faster performance and lower energy consumption among running tasks.

We evaluate our approach using the NIRGAM simulator [6]. We compare our approach against three alternative methods, including a random allocation scheme, INC [5], and CASqA [7]. Experimental results show that our approach is able to reduce the communication by 59.73%, 5.19% and 2.99% and energy consumption by 53.34%, 17.94% and 12.68%, over the random scheme, INC, and CASqA, respectively.

This paper makes the following contributions:

- It is the first to leverage the topology partition theory to model the resource requirement among multiple jobs for NoC.
- It presents a novel heuristic to reduce the resource contention of shared routers among multiple jobs, using the partial topology partition theory.

2 Background

2.1 The Problem of Shared Routers

In this section, a simple example is offered to show the impact of Shared routers on communication latency and energy consumption. Figure 1 shows the results of mapping job1 and job2, to a 5×5 mesh NoC under the XY routing rule. Suppose job1 maps before job2. Figures 1a and 1b show the different results caused by two mapping method. The mapped area and communication distance for each job is the same. However, Fig. 1a has more shared routers than Fig. 1b, where the two blue routers in the red area are the shared routers.

L_a represents the average actual communication latency of job1 and job2 in Fig. 1a. L_b represents it in Fig. 1b. Accordingly, E_a and E_b respectively represent the energy consumed by all routers and their links occupied by jobs in Fig. 1a

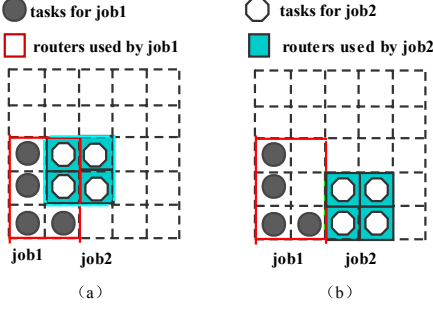


Fig. 1. The results of the two job mapping methods. (a) result with shared routers, and (b) result without shared router

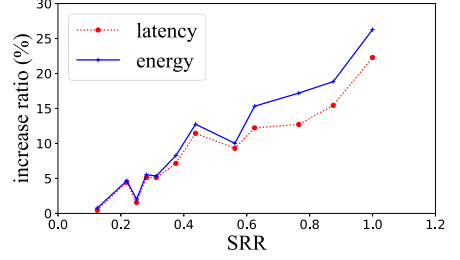


Fig. 2. $r_{latency}$ and r_{energy} change with SRR (Color figure online)

and 1b. Compared with Fig. 1b, communication latency increases by 3.14% and energy consumption increases by 3.81% in Fig. 1a. The Shared routers (Fig. 1a) can influence the communication latency and energy consumption.

2.2 Communication Latency Caused by Shared Routers

Furthermore, we quantitatively analyze the rise in communication latency caused by the increase in Shared routers. According to the *SchedulingMethod* [8], a packet containing n flits transfer from s to d , and the latency calculation formula is as follows:

$$T_{pkt_cont}(s, d) = (T_{receive} + T_{handle} + T_{send}) \times R(s, d) + T_{transfer} \times (n - 1) \times MD(s, d) + T_{handle} \times M \times R(s, d)_{shared} \quad (1)$$

This includes the time it takes for $R(s, d)$ routers to receive, handle and send header flits from s to d (the first item in formula 1), the transfer time of the remaining flits at $MD(s, d)$ communication distance (the second item in formula 1), and the time it takes $R(s, d)_{shared}$ routers to handle M packets in FIFO queues (the third item in formula 1). For Fig. 1b, the third item in formula 1 is 0, because two jobs do not share the router. However, for Fig. 1a, it is not 0 because of the existence of shared routers. Therefore, the communication latency in Fig. 1a is higher than it in Fig. 1b.

Next, we quantitatively analyze the variation of the communication latency under different numbers of shared routers, which are generated by different mapping methods. The number of Shared routers is measured through the shared router ratio (SRR). Random mapping method is used to allocate resources for two jobs in an 8×8 NoC. We get different SRR and sort them in ascending order. The red line in the Fig. 2 shows how the average actual communication latency increments Δ_L changes as SRR increases.

$$\Delta_L = \frac{L_a - L_b}{L_b} \times 100\% \quad (2)$$

It can be seen that with the increase of SSR, Δ_L increases significantly. According to Formula (1), the third item will rise when SRR increase. When the SRR is 0.2, $\Delta_L = 4.46\%$, when SRR increases to 0.87, $\Delta_L = 13.44\%$.

2.3 Communication Energy Caused by Shared Routers

The shared routers will not only impact communication latency but will also, increase the energy consumption of the NoC. The energy consumption of router buffering data increases because of Shared routers in Fig. 1a. That means E_{bf} increases in Formula 3.

A message contains N packets and the size of one packet is L_{pk} bits. It transfers from processor s to processor d . The communication energy E_{com} is calculated by the Formula 3 [8].

$$E_{com}(s, d) = [(E_{xbar} + m \times E_{bf}) \times R(s, d) \times N + E_{c \rightarrow r} + E_{r \rightarrow r} \times (R(s, d) - 1) + E_{r \rightarrow c}] \times L_{pk} \times N + E_{handle} \times R(s, d) \times N \quad (3)$$

It contains three terms. The first item is the energy consumed by N routers. E_{xbar} is the average energy to transfer a bit through a crossbar. E_{bf} is the average energy for buffering a bit. The second item is the energy consumed by the link. $E_{c \rightarrow r}$ and $E_{r \rightarrow c}$ respectively represent the transmission energy from the source core to the direct router, and from the last router to the destination core. $E_{r \rightarrow r}$ represents the average energy to transfer a bit through an electrical interconnect between routers. The third item represents the energy consumption for the router to make decisions for a packet. E_{handle} is the energy of the router to handle header flits. When there are Shared routers, the energy consumed to buffer packets increases due to network contention, that is the part of E_{bf} in Formula 3.

We further quantitatively analyze the variation of the energy consumption increment Δ_E in the case of the different number of Shared routers. The blue line in Fig. 2 shows how the energy increment Δ_E consumed by all the occupied routers and links changes as SRR increases.

$$\Delta_E = \frac{E_a - E_b}{E_b} \times 100\% \quad (4)$$

It can be seen that with the increase of SSR, Δ_E increases significantly. When the SRR is 0.2, $\Delta_E = 4.66\%$, when SRR increases to 0.87, $\Delta_E = 18.83\%$.

Different mapping methods can produce different numbers of shared routers. How to design the mapping method that products as few shared routers as possible is the concern in this paper. At present, most mapping strategies usually allocate resource according to the idle cores and often ignore the real-time occupation of routers. So the situation with shared routers in Fig. 1a is easy to happen. Besides, it is inevitable for routers to be shared by multiple jobs because of the large number of jobs, the limited cores, and the fragmentation in allocation. To solve this problem, the mapping strategy must be reconsidered. The utilization of routers should be one of the crucial conditions for job mapping.

Here are the challenges: How to characterize the occupancy of routers on the chip? How to keep the number of Shared routers as small as possible?

Here are our solutions: Topology partition theory is used to depict routers for each job as well as the shared routers among multiple jobs. A heuristic algorithm based on topology partition is designed to reduce the number of shared routers.

3 Mapping Algorithm Based on Topology Partition

Suppose that a $N \times N$ 2D Mesh structure is designed for multi-core processor. There are already k jobs in the system, noted with JM . The $k + 1$ job J_k is mapped on the NoC at t_0 . Our job mapping algorithm based on topology partition is used to allocate resources for J_k . The algorithm is divided into two parts: core allocation and core mapping. Core allocation is to find a region satisfying the conditions for J_k , that is, to obtain a set of core C_{J_k} . Core mapping implements the one-to-one mapping of processes in J_k to cores in C_{J_k} .

3.1 Examples of Core Region Selection

Here is an example to show the basic idea of region selection. There are 4 ordered jobs, J_1, J_2, J_3, J_4 . The number of processes is $n_1 = 4, n_2 = 6, n_3 = 3, n_4 = 6$, respectively. They will be mapped in a 5×5 NoC. Figure 3 is the selected region for this group of jobs under our mapping method.

A bidirectional balanced mapping based on application size is used to guide the selection for a job: a small job seeks an appropriate area according to ascending order of idle routers. Instead, a large job according to descending order. For a $N \times N$ NoC, this paper takes $n_{th} = N$ as the boundary to distinguish a job, that is, if $n_{job} > n_{th}$, it is denoted as a large job, if not, it is denoted as a small job. For 5×5 NoC, n_{th} is 5.

Figure 3a shows the region selected for J_1 . Since $n_1 < 5$, select the region from the minimum idle router R_1 . Start with R_1 and seek for a rectangular region with four idle cores (square is optimum). R_1 is the top left vertex. Once found, the cores in the region are got, which are c_1, c_2, c_6, c_7 . Figure 3b shows the region selected for J_2 and now J_1 is a part of JM . Since $n_2 > 5$, select the region from the maximum idle router R_{24} . Start with R_{24} and seek for a rectangular region with six idle cores (square is optimum). Then get the cores number in the region $c_{24}, c_{23}, c_{19}, c_{18}, c_{14}, c_{13}$. Figure 3c shows the region selected for J_3 , Since $n_3 = 3$, Start from the minimum idle router and seek for a rectangular region with three idle cores (square is optimum). Then get c_3, c_4, c_8, c_9 . Do the same steps for J_4 and select the region as Fig. 3d.

The use of a specific router is determined by routing rules and communication between processes. For example, the region selected for J_3 is c_3, c_4, c_8, c_9 , but actually J_3 only needs 3 cores. Therefore, its communication should be considered during the core mapping, and *CoreMapping()* in Algorithm 1 is used to realize the mapping of job process to core in the selected region. Finally c_3, c_4, c_8 are selected for processes for J_3 .

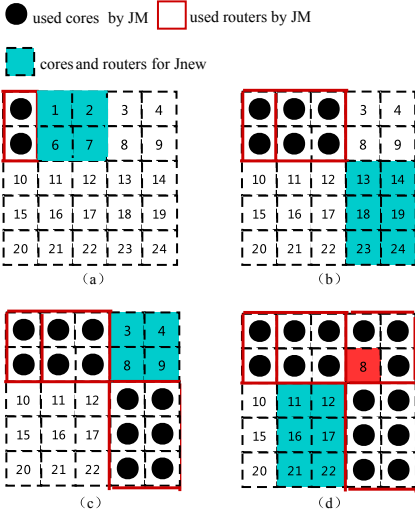


Fig. 3. The result of core allocation for J_1, J_2, J_3, J_4 . (a) core allocation for J_1 ; (b) core allocation for J_2 ; (c) core allocation for J_3 ; (d) core allocation for J_4

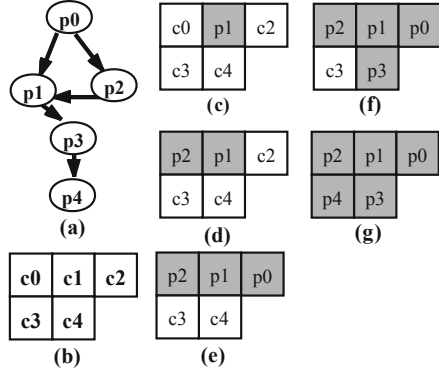


Fig. 4. Core Mapping for *job*. (a) communication graph for *job*; (b) The selected mapping region; (c) map p_1 to c_1 ; (d) map p_2 to c_0 ; (e) map p_0 to c_2 ; (f) map p_3 to c_4 ; (g) mapping results for all processes

3.2 Single Job Mapping Algorithm Based on Topology Partition

The job mapping algorithm based on topology partition is shown in Algorithm 1. The input includes state information of current NoC (used processor cores C_{used} , unused processor cores C_{unused} , routing rule), job information (the number of processes n , the process ordered set based on the total communication volume P_{comm}) and threshold to distinguish jobs- n_{th} . The output consists of a set of cores assigned to $J_{new}-C_{J_{new}}$ and the corresponding relationship between the job process and core-MAP. In step 1, *FindUsedRouters* gets the available routers R_{unused} by routing rules and the used core. R_{unused} , an ascending sequence sorted by the number, is used in *CoreAllocation* to select a set of processor core whose region is a rectangle or close to a rectangle. Steps 2 to 4 determine the order of the traversal of *CoreAllocation* according to the number of processes. For a large job, reverse the sequence, that finds the region from a large number of the router. The fifth step is to call the algorithm *CoreAllocation*, and select the mapping region for J_{new} . The optimal allocation is the minimum rectangular region containing n processor cores, and the output of *CoreAllocation* is the set of processor cores in the selected region. According to the communication relationship between processes and the connection relationship between cores on NoC, *CoreMapping* gets the specific mapping between process and core $MAP = \{p_i \leftarrow c_j | p_i \in P_{comm}, c_j \in C_{J_{new}}, 0 \leq i < n, 0 \leq j < n\}$.

Algorithm 1. Single job mapping algorithm based on topology partition**Require:**

used processor cores C_{used} ; unused processor cores C_{unused} ;
 routing Rule $routing = XYrouting$;
 process numbers of J_{new} n
 the process ordered set based on the total communication volume
 $P_{comm} = \langle p_0, p_1, \dots, p_n \rangle$;
 threshold to distinguish jobs n_{th} ;

Ensure:

A set of processor cores assigned to J_{new} $C_{J_{new}}$;
 The corresponding relationship between the job process and $C_{J_{new}}$
 $MAP = \{p_i \leftarrow c_j | p_i \in P_{comm}, c_j \in C_{J_{new}}, 0 \leq i < n, 0 \leq j < n\}$.
 1: $R_{unused} = FindUsedRouters(G, C_{used}, routing)$; // R_{unused} is an ascending
 sequence sorted by the idle router number
 2: **if** $n > n_{th}$ **then**
 3: $R_{unused} = Reverse(R_{unused})$; // Reverse the sequence R_{unused}
 4: **end if**
 5: $C_{J_{new}} = CoreAllocation(n, C_{unused}, R_{unused})$; // Core allocation algorithm, get a
 set of processor cores
 6: $MAP = CoreMapping(P_{comm}, C_{J_{new}})$; // Core Mapping algorithm, map each
 process to the corresponding core

CoreAllocation: As shown in Algorithm 2, $C_{J_{new}}$ is obtained according to the number of J_{new} 's process n , the router sequence R_{unused} , and the idle cores C_{unused} . In step 1–14, search for the smallest rectangular area containing m cores first ($m \leq n$). Through *GetRectangle* in step 3, obtain the rectangular region containing m cores with router *vertex* as the top-left vertex. If it can be found, return the cores in the region C_{rect} ; If not, use the next *vertex* in R_{unused} to get the region. If the final returned rectangle contains less than n cores, an additional $n - m$ cores are still needed to meet the assignment requirement of J_{new} . Steps 12–22 are the steps to find them. The basic idea is to find the other $n - m$ cores closest to C_{part} (the found region with m cores). This $n - m$ idle cores are searched one by one through the while loop (step 15–21), and then added to C_{part} . Since the number of idle cores is assumed greater than or equal to n , the $n - m$ idle cores can be found when the loop is over. In step 17, select the idle core, which is closest to C_{part} and has the fewest unused neighbours around by *MinmdAndneighbor*. The neighbour here means the core with a Manhattan Distance of 1.

CoreMapping: In this paper, core mapping is based on the algorithm in Chou [5]. The process of mapping each process to the core is divided into two steps. First, an unmapped process is selected in p_{comm} . Then a suitable on-chip core is selected for it. But it's different between our method and Chou's in process selection: Chou defines 3 states of the process, *white*, *gray*, and *black*, two actions to switch between states: *DISCOVER* and *FINISH*. If the neighbors (the processes that communicate with each other are neighbors) of the process p are all *white*, *DISCOVER* it and select all available cores on the slice and convert p to *gray*; If p 's neighbour is *gray* or *black*, *FINISH* it and select a particular core and convert p to *black*. Go back to the first node of the ordered set, change the

state for nonblack process until they all *black*. For it takes two steps for a process mapping to a core, we get rid of *gray*. To reduce the communication distance among processes with high traffic, we strengthen condition for the *FINISH* action. The basic idea is as follows: start by selecting the process with the largest traffic volume to map, and mark it *black*. If the neighbour process with the largest traffic of process p is *white*, choose p to be p_{next} , the next process that needs to be mapped. And p is $p_{neighbor}$, the core for p is $c_{neighbor}$. The process with the most traffic has already been mapped, so such a p_{next} can certainly be found. Select a specific core for p_{next} such that the distance between p_{next} and $c_{neighbor}$ is minimized. If more than one core gets the minimum distance closest to p_{next} , we choose the core that the number of whose neighbors closest to the number of nonblack neighbors of p_{next} .

Algorithm 2. *CoreAllocation*

Require:

the number of J_{new} 's process n ;
the router sequence R_{unused} ;
the idle cores C_{unused} ;

Ensure:

A set of processor cores assigned to J_{new} $C_{J_{new}}$;

```

1: for  $m=n, n-1, n-2, \dots, 1$  do
2:   //Look for a rectangle with  $n$  points, if not, look for a rectangle with  $n-1$ 
   points, and so on.
3:    $C_{part} = \emptyset$ ;
4:   for each vertex in  $R_{unused}$  do
5:      $C_{rect} = GetRectangle(vertex, m)$ ;
       //Return a rectangle with  $m$ -points with  $vertex$  as the vertex
6:     if ( $C_{rect} \neq \emptyset$ ) then
7:        $C_{part} = C_{rect}$ ;
8:       return;
9:     end if
10:  end for
11: end for
12: if (the rectangle contains less than  $n$  cores then
13:   //The size of the rectangle is less than  $n$ , so still need to find an additional
    $n-m$  cores
14:    $C_{unused} = C_{unused} - C_{part}$ ;
15:   while  $m < n$  do
16:     //seek for the other  $n-m$  cores one by one. The rule is to look for other idle
       cores closest to  $C_{part}$ 
17:      $c = MinmdAndneighbor(C_{unused}, C_{part})$ ; //select the idle core in  $C_{unused}$ ,
       which is closest to  $C_{part}$  and has the fewest unused neighbors
18:     join  $c$  to  $C_{part}$ ;
19:     remove  $c$  from  $C_{unused}$ ;
20:      $m = m + 1$ ;
21:   end while
22: end if
23:  $C_{J_{new}} = C_{part}$ ;
24: END

```

As shown in Fig. 4, job with five processes in (a) is mapped to the selected region shown in (b). (c), (d), (e), (f) and (g) are its mapping processes. Assume now that, based on the total communication volume, the process ordered set is

$p_{comm} = \langle p_1, p_2, p_0, p_3, p_4 \rangle$. We start with p_1 , since it has the largest communication volume. And it is mapped to c_1 who has the most neighbors, as shown in (c), $p_1 \leftarrow c_1$ is joined to MAP . At this time, p_2 , the process that have the most traffic with p_1 , is chosen as p_{next} . $p_{neighbor}$ is p_1 , $c_{neighbor}$ is c_1 . The unmapped process that communicate with p_2 is p_0 . Select the core closest to c_1 and the available neighbor is 1. c_0 and c_4 are both meet the requirements. Select the one with the smaller number, and as shown in (d), $p_2 \leftarrow c_0$ is added to MAP . Follow this step to get $p_0 \leftarrow c_2$, $p_3 \leftarrow c_4$, $p_4 \leftarrow c_3$, and the mapping result is shown in (e).

3.3 Computation Complex

CoreAllocation: A job with n processes is allocated to $N \times N$ NOC, where $n \leq N^2$. Scanning the R_{unused} list executes $|n|$ times. For each router in R_{unused} , *GetRectangle* and while loop (in line 15) executes $|n|$ times. So the total run time for *CoreAllocation* has a complexity of $O(n^2)$.

CoreMapping: An ACG for a job with n processes and e edges is mapped to the selected region. The total run time of our algorithm has a complexity of $O(n^2 + e)$ the same with Chou [5].

4 Experimental Results

4.1 Experimental Platform

Simulation Environment. In this paper, NIRGAM [6] is used to simulate an 8×8 mesh NoC. Table 1 is the configuration of NIRGAM in this experiment:

Table 1. Configuration for NIRGAM platform

Parameter name	Values	Description
TOPOLOGY	MESH	2-d mesh topology
NUM ROWS	8	8 rows
NUM COLS	8	8 columns
RT ALGO	XY	XY routing algorithm
NUM BUFS	16	The number of buffers in input channel FIFO is 16
CLK FREQ	1 GHz	Clock frequency is 1 GHz
PKT SIZE	32	Packet size is 32 bytes
FLIT INTERVAL	1	Interval between successive flits is 1 clock

Job Sequence Generation. Several sets of jobs with 4 to 16 tasks are generated using the TGFF [9]. It's 4 to 8 for small-scale-job, 9 to 16 for large-scale-job. Adjusting the proportion of large jobs to 0%, 25%, 50%, 75% and 100%, we get 5 sets of jobs. An arrival sequence is generated in each set. These sequences are used to simulate the order in which the OS allocates resources for actual jobs. NPB [10] traces with 4 and 8 (9 for BT and SP) and 16 processes are get by HPC-NetSim [11]. An arrival sequence is generated for NPB.

The mapping algorithms we compare include random, INC [5], CASqA [7] and the Job mapping algorithm based on topology partition (JMATD) proposed in this paper. FT2000+ under the condition of not binding cores, allocates resource for jobs in a random way by default. INC is a convex region mapping algorithm, which can reduce communication energy consumption and improve the application's performance. CASqA has multiple mapping levels by adjusting threshold (α), where set $\alpha = 0$ to improve performance and reduce communication energy consumption and latency.

4.2 Experimental Result

The Number of Shared Routers. The number of Shared routers produced by the four algorithms is different. In order to compare the differences, the number of shared routers in the five job sequences is statistically analyzed. (a), (b), (c), (d), (e), (f) in Fig. 5 respectively reflect the change in the number of Shared routers per job sequence during the mapping process. For random, the number of Shared routers is the largest due to the overlap of jobs. JMATD reduces the number of Shared routers by partitioning the topology when a single job is mapped. For each job sequence, JMATD has a good optimization effect. In (d), the number of Shared routers in random, INC and CASqA is 9.56x, 3.48x and 2.10x higher than that in JMATD, respectively. For both INC and CASqA, due to the continuous convex mapping region, the same effect can be achieved with JMATD for the job sequences with more fragment, as shown in (b). Figure 7 shows the average number of Shared routers in the mapping results for each group of job sequences. On average, the number of Shared routers generated by random, INC and CASqA is 5.78x, 1.25x and 0.67x higher than that of JMATD.

Communication Power for Jobs. JMATD is effective in reducing the number of Shared routers. We measured the communication power curve changing under different mapping algorithms for each set of jobs, and the results are shown in Fig. 6(a), (b), (c), (d), (e), (f) represent different job sequences. Although the power curve of each mapping method fluctuates somewhat, JMATD method is relatively low compared with other methods on the whole. Figure 8 shows the average power consumption in the job mapping process. On average, Compared with random, INC, and CASqA, the communication energy consumption is decreased by 53.34%, 17.94%, 12.68%, respectively. Run time is assumed to be the same for a job in different mapping method. Therefore, the communication energy consumption of the job sequence is proportional to the power consumption.

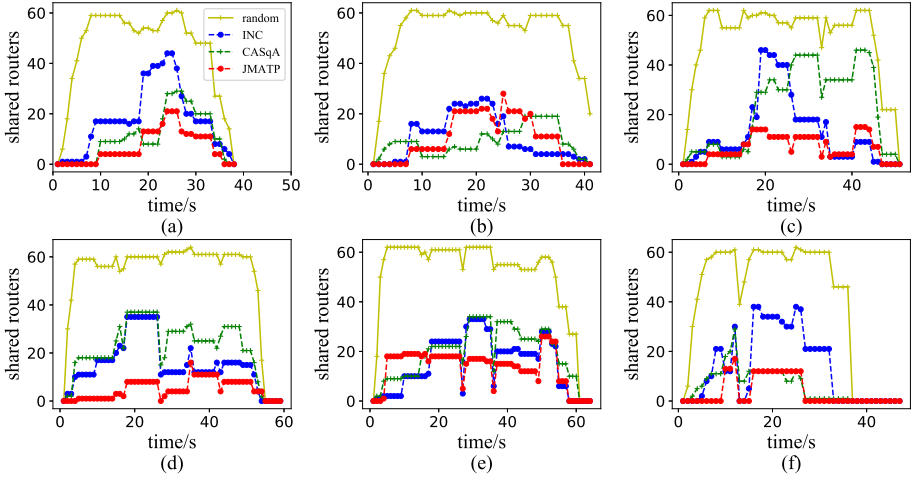


Fig. 5. Changes in the number of Shared routers per job sequence, (lower is better); (a) 1% = 0%; (b) 1% = 25%; (c) 1% = 50%; (d) 1% = 75%; (e) 1% = 100%; (f) NPB

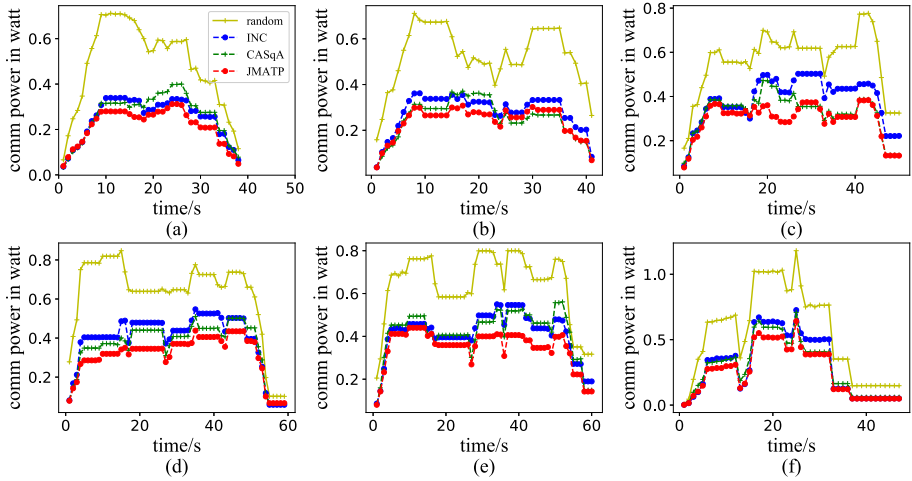


Fig. 6. Changes in communication power during each job sequence mapping, (lower is better); (a) 1% = 0%; (b) 1% = 25%; (c) 1% = 50%; (d) 1% = 75%; (e) 1% = 100%; (f) NPB

Communication Latency for Jobs. The communication latency of a job is an important factor affecting performance. To compare the effect of JMATP in reducing communication latency, the average latency of each job is calculated. As shown in Fig. 9, the data is normalized. According to the average results of the five job sets, the latency of random, INC, CASqA method is 59.73%, 5.19%, 2.99% higher than that of JMATP, respectively.

4.3 Discussion

The experiment shows that the number of Shared routers has an effect on the communication power consumption of the system, and the trend in Fig. 8 is consistent with that in Fig. 7. However, when $l\% = 25\%$, the router is shared equally in INC, CASqA and JM ATP, but the communication power is quite different. JM ATP has lower power. We compared the communication distances of the jobs in each algorithm. In this article, weighted Manhattan distance (WMD, the sum product of MD and the corresponding weight of communicating processes) [12] is a metrics of the quality of the job mapping. Figure 10 is the WMD comparison of 4 mapping methods adopted for 6 sets of jobs. From the perspective of single job mapping, compared with other algorithms, JM ATP can effectively reduce the communication distance between job processes by 63.82%, 22.39% and 19.07% respectively for random, INC and CASqA in WMD. JM ATP not only reduces the number of Shared routers but also reduces the communication distance of jobs.

Formula 1 indicates that the latency is related to the router and the communication distance of the process where the contention occurs. Since the packet transmission is concurrent, the latency is not wholly positive related to the relationship between the two. It means that the marginal benefits of optimizing communication latency from reducing communication distance are not high. Therefore, it is necessary to optimize shared routers while reducing the communication distance.

The influencing factors of communication latency include external congestion (router and link contend by different jobs) and internal congestion (router and link contend by packets of the same job). Memory [13] and disks [14] also impact the communication. Because of the interaction of these factors, communication latency optimization is not significant compared to communication power. It inspires us to work on what we're going to do next: How to reduce resource contention between job processes. How job communication characterizations [15] affect communication latency.

5 Related Work

The performance of NoC is closely related to network congestion, which not only increases network latency but also affects the communication power consumption. That is why there are many works to diminish network congestion from software and hardware aspects. Ebragimi [16] optimizes communication from routing algorithms to reduce network conflicts. Jiang [17] proposed a new switching mechanism to reduce network latency. Based on the STT-RAM router, Yang [18] reduces communication latency by calculating the contended flit.

Job mapping is one of the effective ways to reduce network conflicts. Two types of congestion can be defined during dynamic application mapping: external and internal congestion. External congestion occurs when a network channel is competing with packets from different applications; internal congestion is related to packets from the same application.

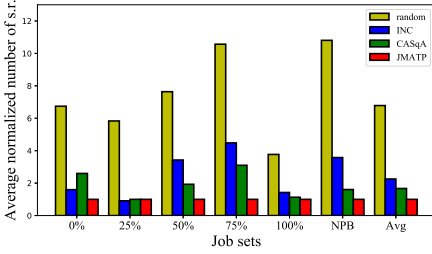


Fig. 7. Average number of Shared routers for jobs (lower is better)

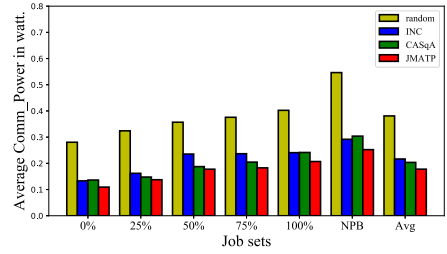


Fig. 8. Average communication power for jobs (lower is better)

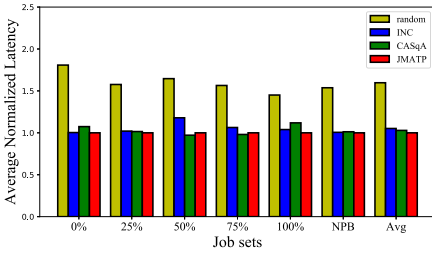


Fig. 9. Average latency for jobs (lower is better)

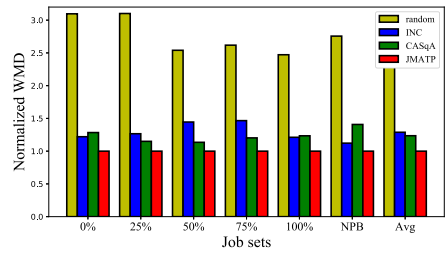


Fig. 10. WMD comparison for jobs (lower is better)

To decrease the external congestion probability by mapping algorithm Chou [5] proposes an incremental (INC) approach. They first select the near convex region to reduce communication links and try to keep both the selected region and remaining nodes contiguous, then allocation node according to the total communication volume inside the selected region. Das [19] proposes new mapping policies to improve system performance by reducing inter-application interference in the on-chip network and memory controllers. Cores are clustered into a subnetwork. Fattah in [12], proposed a SHiC algorithm to guide how to find the optimum first node among all the available nodes for the run-time application. Then, in [7] they proposed a run-time mapping algorithm, CASqA. In this algorithm, the contiguity of the allocated processors can be adjusted in a fine-grained fashion according to α . Zhu [20] proposed an efficient heuristic-based algorithm to balance minimized on-chip latency in multi-application mapping.

Internal congestion can also be reduced by the mapping algorithm. In [8], a heuristic algorithm, unified priority-based scheduling (UPS), is put forward to effectively solve the contention problem in polynomial time by assigning priorities to messages. Once an instruction is waiting for the data from other PE, the extra delay caused by NoC congestion postpone the instruction issue and decrease the performance. An [21] proposed C-Map for the delay of the instructions existing in CGRA, which improves the effectiveness of CGRA mapping in the perspective of reducing network congestion and enhancing the continuity of the data-flow.

6 Conclusion

This paper analyzes the influence of Shared routers among multiple jobs on communication latency and energy consumption. When the number of Shared routers increases significantly, it affects the communication latency of a single job and the energy consumption of NoC. To reduce this impact, this paper proposes a task mapping method based on topology partition. When allocating resources for a single job, cores connected to an idle router are considered first to minimize the number of shared routers between multiple jobs. NIRGAM Simulator is used to compare the mapping method proposed in this paper and three other typical ones (including random, INC, and CASqA). Communication latency and energy consumption of the jobs under each mapping method are get based on an 8×8 NoC. The communication performance is improved to 59.73%, 5.19%, 2.99% and energy consumption is decreased by 53.34%, 17.94%, 12.68%, respectively. Shared routers exist not only between jobs but also between processes in a job. Next, We focus on how to reduce Shared routers in the same application. We also consider the impact of memory and disks on communication.

References

1. Liu, W., et al.: Thermal-aware task mapping on dynamically reconfigurable network-on-chip based multiprocessor system-on-chip. *IEEE Trans. Comput.* **67**(12), 1818–1834 (2018)
2. Kahng, A.B., Li, B., Peh, L., Samadi, K.: Orion 2.0: a fast and accurate NoC power and area model for early-stage design space exploration, pp. 423–428 (2009)
3. Wu, C., et al.: An efficient application mapping approach for the co-optimization of reliability, energy, and performance in reconfigurable NoC architectures. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst.* **34**(8), 1264–1277 (2015)
4. Chen, L., Zhu, D., Pedram, M., Pinkston, T.M.: Power punch: towards non-blocking power-gating of NoC routers, pp. 378–389 (2015)
5. Chou, C., Ogras, U.Y., Marculescu, R.: Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst.* **27**(10), 1866–1879 (2008)
6. NIRGAM: A simulator for NoC interconnect routing and application modeling. <https://nirgam.ecs.soton.ac.uk/>
7. Fattah, M., Liljeberg, P., Plosila, J., Tenhunen, H.: Adjustable contiguity of runtime task allocation in networked many-core systems, pp. 349–354 (2014)
8. Yang, L., Liu, W., Jiang, W., Li, M., Yi, J., Sha, E.H.M.: Application mapping and scheduling for network-on-chip-based multiprocessor system-on-chip with fine-grain communication optimization. *IEEE Trans. Very Large Scale Integr. Syst.* **24**(10), 3027–3040 (2016)
9. TGFF. <http://ziyang.eecs.umich.edu/~dickrp/projects/tgff/index.html>
10. NAS parallel benchmarks. <https://www.nas.nasa.gov/publications/npb.html>
11. Zhou, W., Chen, J., Cui, C., Wang, Q., Dong, D., Tang, Y.: Detailed and clock-driven simulation for HPC interconnection network. *Front. Comput. Sci. China* **10**(5), 797–811 (2016)
12. Fattah, M., Daneshmand, M., Liljeberg, P., Plosila, J.: Smart hill climbing for agile dynamic mapping in many-core systems. In: 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6, May 2013

13. Wu, F., et al.: A holistic energy-efficient approach for a processor-memory system. *Tsinghua Sci. Technol.* **24**(4), 468–483 (2019)
14. Dong, Y., Chen, J., Tang, Y., Wu, J., Wang, H., Zhou, E.: Lazy scheduling based disk energy optimization method. *Tsinghua Sci. Technol.* **25**(2), 203–216 (2019)
15. Chen, J., et al.: Analyzing time-dimension communication characterizations for representative scientific applications on supercomputer systems. *Front. Comput. Sci.* **13**(6), 1228–1242 (2018). <https://doi.org/10.1007/s11704-018-7239-1>
16. Ebrahimi, M., Daneshlab, M., Farahnakian, F.: HARAQ: congestion-aware learning model for highly adaptive routing algorithm in on-chip networks. In: 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, pp. 19–26, May 2012
17. Jiang, G., Li, Z., Wang, F., Wei, S.: A low-latency and low-power hybrid scheme for on-chip networks. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **23**(4), 664–677 (2015)
18. Yang, L., Liu, W., Guan, N., Dutt, N.: Optimal application mapping and scheduling for network-on-chips with computation in STT-RAM based router. *IEEE Trans. Comput.* **68**(8), 1174–1189 (2019)
19. Das, R., Ausavarungnirun, R., Mutlu, O., Kumar, A., Azimi, M.: Application-to-core mapping policies to reduce memory system interference in multi-core systems. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 107–118, February 2013
20. Zhu, D., Chen, L., Yue, S., Pinkston, T.M., Pedram, M.: Balancing on-chip network latency in multi-application mapping for chip-multiprocessors, pp. 872–881 (2014)
21. An, S., et al.: C-map: improving the effectiveness of mapping method for CGRA by reducing NoC congestion. In: High Performance Computing and Communications, pp. 321–328, August 2019



Dissecting the Phytium 2000+ Memory Hierarchy via Microbenchmarking

Wanrong Gao, Jianbin Fang^(✉), Chuanfu Xu, and Chun Huang

College of Computer, National University of Defense Technology, Changsha, China
{gaowanrong, j.fang, xuchuanfu, chunhuang}@nudt.edu.cn

Abstract. An efficient use of the memory system on multi-cores is critical to improving data locality and achieving better program performance. But the hierarchical memory system with caches often works in a “black-box” manner, which automatically moves data across memory layers, and makes code optimization a daunting task. In this article, we dissect the memory system of the Phytium 2000+ many-core with microbenchmarks. We measure the *latency* and *bandwidth* of moving cache-lines across memory levels on a single core or two distinct cores. We design a set of micro-benchmarks by using the *pointer-chasing* method to measure latency, and using the *chunk-accessing* method to measure bandwidth. During measurement, we have to place the cacheline on the specified memory layer and set its initial consistency state. The experimental results on Phytium 2000+ provide a quantified form of its actual memory performance, and reveal undocumented performance data and micro-architectural details. To conclude, our work will provide quantitative guidelines for optimizing the Phytium 2000+ memory accesses.

Keywords: Phytium 2000+ · Memory hierarchy · Microbenchmark

1 Introduction

Compared with single-core processors, multi-core processors have to deal with significantly more concurrent memory accesses [4]. The memory system has thus introduced a multi-level caching hierarchy to “lock” the frequently accessed data, aiming to minimize the accesses to the off-chip memory. Modern cache features, such as the number of cache layers, each layer’s capacity, to use the *inclusive* or *exclusive* policy, and so on, vary across multi-core architectures. In addition, the memory system of modern multi-cores often works in the form of a “black box”, i.e., many implementation details are not disclosed. And the official technical specifications only reveal theoretical numbers and is of little significance in guiding the actual performance engineering. All these bring programmers a huge challenge of optimizing codes on the cache-coherent multi-core architectures. Therefore, it is significant to dissect the working mechanism of multi-core memory systems through quantifying the actual performance behaviours.

Prior works have demonstrated how well the memory hierarchy performs on the conventional multi-core architectures. The **STREAM** benchmarks focus on

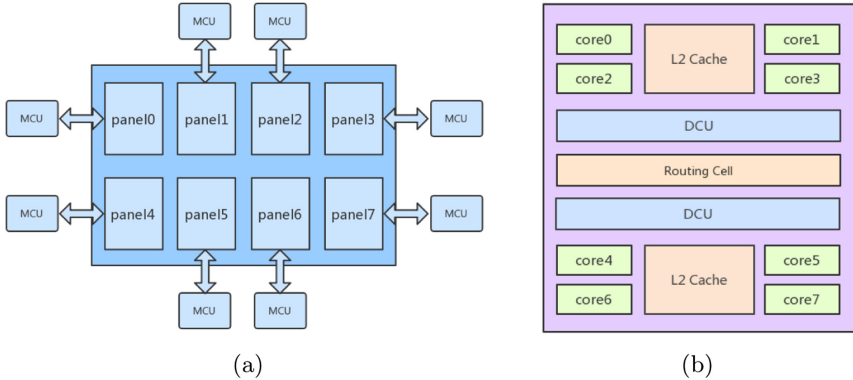


Fig. 1. The Mars II microarchitecture and the panel structure of Phytium 2000+ .

measuring the memory throughput, i.e., data loading/storing bandwidth with a single-core or multi-cores [5]. McVoy and Staelin present a set of micro-benchmark suite (`lmbench`) to quantify the performance of various computer components [6]. In particular, they use *pointer-chasing* to measure the overhead of moving data across cache layers. But `lmbench` ignores the communication overhead of moving cachelines across processing cores. Such performance numbers are essential when optimizing parallel programs concerning shared memory accesses, producer-consumer or thread migration. For this, Molka *et al.* provide a set of microkernels (`BenchIT`) to characterize memory systems [7]. But `BenchIT` is only applicable to the x86 architecture and its memory hierarchy.

In this work, we dissect the memory hierarchy and quantify the achievable performance on Phytium 2000+ (an ARMv8-based cache-coherent 64-core architecture). We measure the communication performance of moving cachelines between distinct cores in terms of *latency* and *bandwidth*, through microbenchmarking (Sect. 3). We obtain undisclosed performance data and reveal many micro-architecture details of Phytium 2000+ on both bandwidth (Sect. 4) and latency (Sect. 5). Our evaluation results provide a quantitative reference for analyzing, modelling, and optimizing the performance of parallel codes on multi-core processors. To the best of our knowledge, this is the first effort of dissecting the memory hierarchy of the Phytium 2000+ architecture.

2 Phytium 2000+ and Its Memory Hierarchy

Phytium 2000+ uses the Mars II architecture [8]. Figure 1(a) gives a high-level view of the Phytium 2000+ processor. It features 64 high-performance ARMv8 compatible processing cores. These cores are organized into 8 panels, where each panel connects a memory control unit (MCU).

The panel architecture is shown in Fig. 1(b). Each panel has eight Xiaomi cores, and each core has a private L1 cache of 32 KB for data and instructions, respectively. Every four cores form a **core group** and share a 2 MB L2 cache.

Note that, the L2 cache of **Phytium 2000+** uses a *inclusive* policy, i.e., the data cachelines stored in the L1 cache are also present in the L2 cache.

Each panel contains two Directory Control Units (DCU) and one **routing cell**. The DCUs on each panel act as dictionary nodes of the entire on-chip network. With these function modules, **Mars II** conducts a hierarchical on-chip network, with a local interconnect on each panel and a global connect for the entire chip. The former couples cores and L2 cache slices as a local cluster, achieving a good data locality and short communication distance. The latter is implemented by a configurable cell-network to connect panels to gain a better scalability. **Phytium 2000+** uses a home-grown **Hawk** cache coherency protocol to implement a distributed directory-based global cache coherency across all panels.

3 Our Approach

This section introduces the design and implementation details of our benchmarks to measure the bandwidth and latency of the **Phytium 2000+** memory hierarchy.

3.1 Benchmarks Design

We measure the sustainable bandwidth by continuously accessing a chunk of data elements, which is shown in Fig. 2(a). In contrast, we use *pointer-chasing* to measure the latency of loading a cacheline by randomly accessing discontinuous data elements (Fig. 2(b)). In this way, we aim to mitigate the impact of hardware and/or software prefetching.

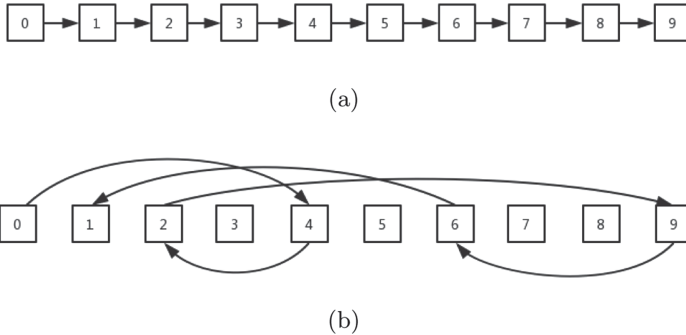


Fig. 2. The data accessing schemes for measuring bandwidth and latency: (a) accessing contiguous data elements to measure bandwidth, and (b) accessing randomly linked data elements to measure latency.

During the measurement, we use multiple threads to move data between cores. To ensure that the buffer allocated by a thread belongs to a fixed core, we pin each thread to a fixed core, i.e., thread n always runs on core n (**cn**).

Besides, we have to control coherency states (**modified**, **exclusive**, **shared**) of cachelines. We use the methods stated in [7], to set the initial coherency state. To determine which level the data is suited in, we control the size of the input datasets to be accessed. And we use a cache flush routine to replace the data in this cacheline with dummy data to evict the measurement data to the next-level cache. The benchmarking steps of measuring latency are shown in Algorithm 1. Here we assume that `c0` loads data from `cn`. The steps of measuring bandwidth are the same, except the way of preparing the initial data.

Algorithm 1. The benchmarking steps of measuring latency

Require: $n \geq 0$

```

1: for  $t = 1$  to  $n$  do
2:   initialize a thread  $Thread_t$ 
3:   cpu_set(mem_bind[t])
4:    $thread[t].status \leftarrow WAIT$ 
5: end for
6: for  $t = 0$  to  $n$  do
7:   // Prepare data and set the initial coherency state of the cacheline
8:   if  $n == 0$  then
9:     // Access local caches
10:    prepare_memory(thread[t])
11:   else
12:     // Access caches on other cores
13:     $thread[t].status \leftarrow PREPARE\_MEMORY$ 
14:    prepare_memory(thread[t])
15:     $thread[t].status \leftarrow DONE$ 
16:   end if
17:   // The cache flush routine
18:   if  $n == 0$  then
19:    flush_caches(thread[t])
20:   else
21:     $thread[t].status \leftarrow FLUSH$ 
22:    flush_caches(thread[t])
23:     $thread[t].status \leftarrow DONE$ 
24:   end if
25:   // Measurement
26:   use assembly instruction to access data
27: end for

```

3.2 Benchmarks Implementation

When implementing benchmarks on the Phytium 2000+ processor, we have to address the following architecture-specific details.

Enabling the Clock-Wise Timing. Our benchmarks are designed to measure the performance of the Phytium 2000+ memory system. For such an measurement, we need a clock-wise timer. It is straightforward to do so by using the `rdtsc` instruction to read the timestamp on the `x86` architecture. Similarly, we can enable the clock-wise timing with the Performance Monitors Cycle Count Register (`PMCCNTR_ELO`) on the ARMv8-based architecture. But this register is only accessible in the kernel mode. To address the issue, we use a kernel module to activate the performance monitoring unit. The key steps of this kernel module are summarized as follows.

- Reading the contents of the control register `PMCR_ELO`.
- Activating the user mode by writing `PMUSERENR_ELO`.
- Resetting all hardware counters by writing `PMCR_ELO`.
- Enabling the performance counter by writing `PMCNTENSET_ELO`.

With this kernel module, the `PMCCNTR_ELO` register can be accessible through the `mrs` instruction to obtain the starting and ending timestamps.

Using the Vector Instructions. To obtain the maximum bandwidth, we have to use the vector instructions to read/write data from/to the memory system. The ARMv8-based architecture extends NEON with 32 128-bit vector registers, while keeping the use of the same mnemonics as general registers [1]. The vector instructions are thus supported on the Phytium 2000+ processor. In the implementation of its SIMD instruction, registers can hold one or more elements of the same size and type. In assembly instructions, the register can identify the vector format including `Vn` (128-bit scalar), `Vn (.2D, .4S, .8H, .16B)` (128-bit vector) and `Vn (.1D, .2S, .4H, .8B)` (64-bit vector). When moving data between registers and memory, we use the `LD1/ST1` instruction of the ARMv8 architecture, similar to `movqda` on the `x86` architecture. The selected vector format is 4 single-precision floating-point words (`.4S`).

Using Special Instructions. Beside the general instructions, we use special instructions shown in Table 1. `DC CIVAC` is used to invalidate specified cachelines. It is useful when controlling the initial coherency state of cachelines. To put target data into the right cache space, we use `DMB` to ensure that the Phytium 2000+ processor does not optimize the execution order of the fetch instructions. In addition, we use the `ALIGN` instruction to avoid unaligned memory accesses.

Table 1. The special ARMv8 instructions [10].

<code>DC CIVAC</code>	Data or unified cacheline clean and invalidate by VA to PoC
<code>DMB</code>	Data memory barrier acts as a memory barrier that Explicitly enables the exeuction of memory access instructions in front of it
<code>ALIGN</code>	Align instruction or data storage address

Table 2. c0 read bandwidth (GB/s).

	Exclusive		Modified		Shared		RAM
	L1	L2	L1	L2	L1	L2	
c0	33.6	18.5	33.6	18.5	33.6	18.5	6
c1	13.3		13.3		18.5		
c4	10.5	10.9	10.5	10.9	10.9		
c8	9.2	9.7	9.2	9.7	9.3		

4 Bandwidth Results

In this section, we measure the read bandwidth on the Phytium 2000+ architecture. Figure 3 show the bandwidth of c0 loading cachelines which are **exclusive**, **modified**, or **shared** in different cores and different cache levels. Table 2 gives a high-level view of the bandwidth numbers. We measure the bandwidth of c0 loading data from its local cache, from c1 sharing a L2 cache with c0, from c4 on the same panel, and from c8 on a different panel.

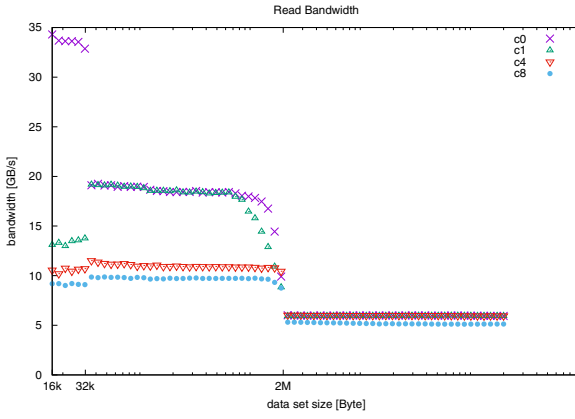
In Fig. 3 we find that the read bandwidth results show a clear phase change as the size of the data set increases. Moreover, the size of the data set when the staged change occurs is basically consistent with the size of various levels of cache. Compared with the first change occurring exactly at 32k (size of L1 cache), the second change occurs earlier than 2M (size of L2 cache). This is because L1 cache is a data cache explicitly, while L2 cache is a hybrid cache including data and instructions both.

Local Accesses. Whatever the state of the cachelines, data can be loaded from c0’s local caches. The obtained bandwidth has nothing to do with the coherency state of the accessed data. The read bandwidth to its local L1 cache can reach 33.6 GB/s, while reading data from the local L2 cache can reach a bandwidth of 18.5 GB/s. Given that the L1 read port of Phytium 2000+ is 128 bits in width and runs at 2.2 GHz, we calculate the theoretical L1 read bandwidth as

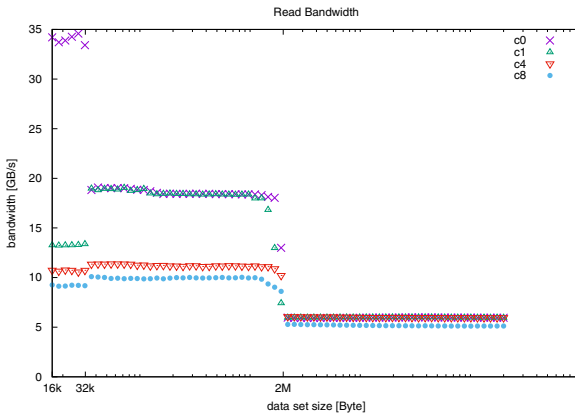
$$2.2 \times 128 \div 8 = 35.2 \text{ GB/s} \quad (1)$$

We see that the measured bandwidth is close to its theoretical counterpart (33.6 GB/s vs. 35.2 GB/s). The measured write bandwidth stays about 17.4 GB/s for L1. We note that the write bandwidth is around a half of the read bandwidth. This is because storing data into L1 occurs at 64 bits per cycle.

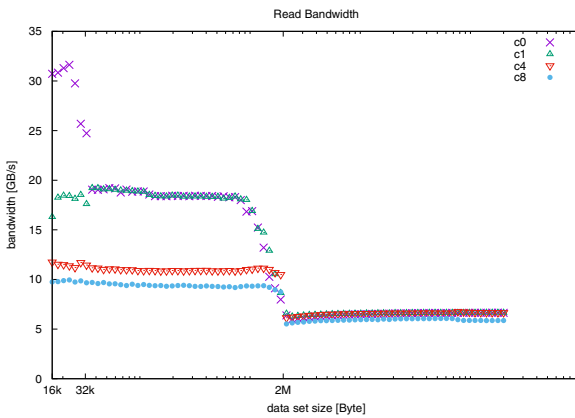
Within a Core Group. Given that c1 and c0 shares the same L2 cache slice, data can be loaded from the local L2 cache when the cacheline is **shared**. And the memory bandwidth of accessing the local L2 can reach 18.5 GB/s. The bandwidth stays the same when cachelines are **exclusive** or **modified** and suited only in the local L2. But the bandwidth is reduced to be around 13.3 GB/s when c0 loading **exclusive** or **modified** cachelines suited in c1’s local L1 cache.



(a) Exclusive



(b) Modified



(c) Shared

Fig. 3. Read bandwidth of c0 accessing the local or another core (c1, c4 or c8)

This is a notable difference from the `x86` processor that the data can be loaded directly from the shared cache slice, when the cacheline is `exclusive` initially. Only when the cacheline is `modified`, the data has to be loaded from the remote higher level cache. But on the Phytium 2000+ processor, we observe that data have to be loaded from a higher cache level for both `exclusive` and `modified` cachelines.

Within a Panel. When `c0` loads data from `c4` of the same panel, where the two cores share no common cache slices, the bandwidth will be limited by cross-group links. As can be seen from Fig. 3, the bandwidth is significantly smaller (by around 40%) than the case when sharing the same L2 cache slice.

Similarly to `c1`, when performing cross-group access to `c4` for `exclusive` or `modified` cachelines, the bandwidth for reading the remote L1 cache is always smaller than accessing the remote L2. This is also because data can be obtained directly from the L2 cache only when its state is `shared` initially.

Across Panels. `c8` does not share a common L2 cache slice with `c0`, and the two cores have to be communicated via the cross-panel routing cells. The read bandwidth of `c0` accessing `c8` ranges from 9.2 GB/s to 9.7 GB/s, which is smaller than the bandwidth of accessing `c1` or `c4` within a panel.

Memory. Since `c0`, `c1`, `c4` are within the same panel, they are connected directly to the same MCU and memory module. When accessing data in the local memory module for `c1` and `c4`, the bandwidth can reach around 6 GB/s. On the other hand, `c8` is connected directly to another MCU and memory module. The bandwidth of `c0` loading data from `c8`'s memory module is around 5.1 GB/s.

To summarize, there is another difference between the Phytium 2000+ processor and the `x86` processor when accessing the `shared` cachelines. The `x86` processor uses an extension of the MESIF protocol, which requires data to be fetched from the core with the latest copy (`forward`). Meanwhile, the Phytium 2000+ processor uses a MOSEI-like coherency protocol. There is no need to find the `forward` copy, but it can directly obtain the data with an arbitrary `shared` copy.

5 Latency Results

This section shows the latency for the Phytium 2000+ memory hierarchy. The performance numbers are measured when the cachelines are `modified` initially.

5.1 Overview of the Latency Results

Figure 4 shows the latency results when `c0` loading data from its local cache, from `c1` sharing a L2 cache slide with `c0`, from `c4` on the same panel, and from `c8` on a different panel. Table 3 shows an overview of the measured latency results.

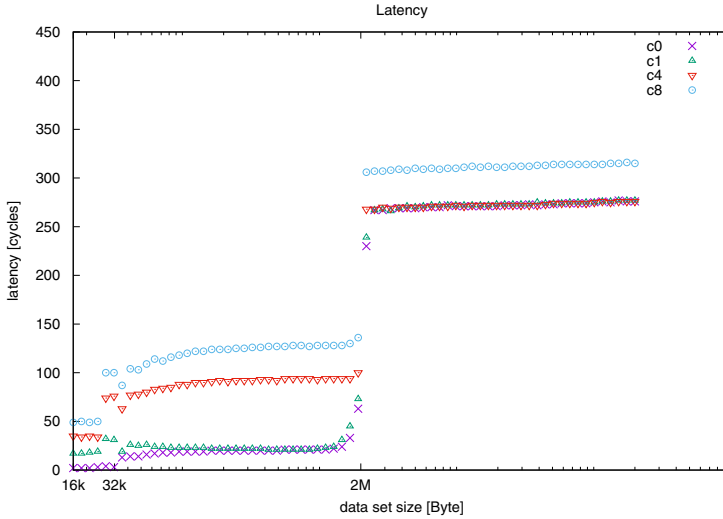


Fig. 4. Read latency of c0 accessing the local (c0) or another core (c1, c4 or c8).

Table 3. c0 read latency (cycle (ns)).

	L1	L2	RAM
c0	3 (1.4)	21 (9.5)	271 (123.2)
c1	18 (8.2)		
c4	34 (15.5)	94 (42.7)	
c8	49 (22.3)	127 (57.7)	

We see that, the latency of accessing the local L1 and L2 cache are 3 cycles (1.4 ns) and 21 cycles (9.5 ns), respectively. For the **Mars II** architecture, there is no public specification documenting such numbers. The specification of the first generation **Mars** describes that accessing the local L1 and L2 takes 2 ns and 8 ns, respectively, which is in accordance with our measured results [11]. When c0 loading data from c1, the latency is same as accessing the local L2 cache. Figure 4 shows that, no matter which memory layer the data is in, loading cachelines across core groups or panels takes many more cycles than accessing the local cache slices. Thus, loading data within a core group is the fastest.

5.2 Across-Panel Latency Results

We evaluate the performance impact of panel distance on latency when accessing cores fixed to different panels. Figure 5 shows the latency results when c0 accessing the cores on p1 (panel 1)–p7 (panel 7), respectively.

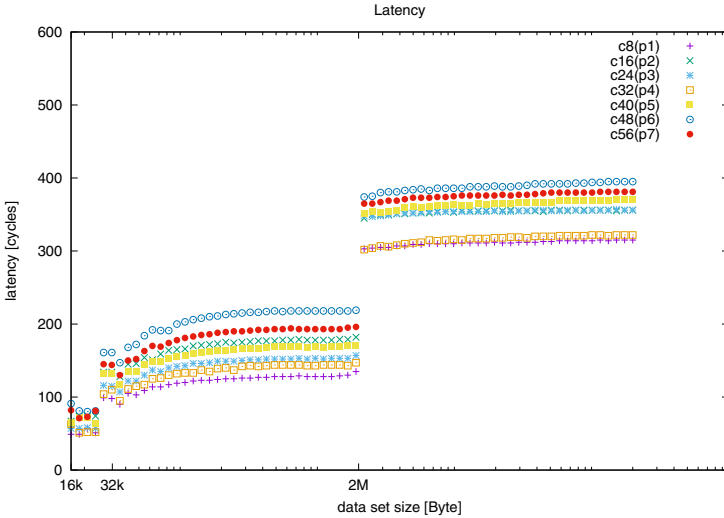


Fig. 5. Read latency of c0 accessing p1–p7.

We see that the latency numbers vary over the panel distance, with the latency difference of up to 105 cycles. Besides, the latency numbers of c0 on p0 accessing c8 on p1 and c32 on p4 are the same. This is because p1 and p4 are at the same distance to p0 (Fig. 1). This result also agrees with the theoretical latency results [11].

5.3 With Different Page Sizes

We investigate the performance impact of the TLB page size on latency. Figure 6 shows the latency measured with the 4KB page, and the other configurations stay the same as that for Fig. 4.

Phytium 2000+ provides the usage of 4KB and 2MB pages. With 2MB page (Fig. 4), the latency of each cache level looks stable. While using the 4KB pages (Fig. 6), although the gap between different cache levels is still visible, the latency increases over the amount of data being accessed. This is because the latency measurement uses the pointer-chasing method. In the data preparation stage, the next-to-be accessed address is randomly generated, resulting in a poor locality for the linked-list access. When using small pages, there will generate too many page table entries, leading to frequent TLB misses and resulting in a large memory access overhead. The bandwidth measurement does not have this issue because its access is consecutive.

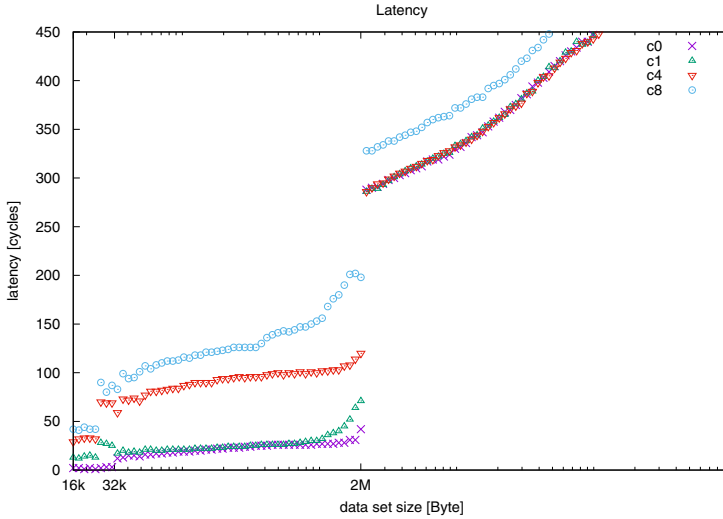


Fig. 6. Read latency of c0 accessing the local (c0) or another core (c1, c4 or c8) using 4 KB pages.

6 Related Work

Although the effective use of the memory systems is essential to obtain the best performance, vendors seldom provide the details of the memory hierarchy or the achieved performance. For this reason, researchers have to obtain such performance results and implementation details through measurements.

Babka *et al.* [2] propose experiments that investigate detailed parameters of the x86 processors. The experiment is built on a general benchmark framework and obtains the required memory parameters by performing one or a combination of multiple open-source benchmarks. It focuses on detailed parameters including the address translation miss penalties, the parameters of the additional translation caches, the cacheline size, and the cache miss penalties.

McCalpin *et al.* [5] present four benchmark kernels (Copy, Scale, Add, and Triad), **STREAM**, to access memory bandwidth for a large variety current computers, including uniprocessors, vector processors, shared-memory systems, and distributed-memory systems. **STREAM** is one of the most commonly used memory bandwidth measurement tools in Fortran and C. But it focuses on throughput measurement without considering the latency metric.

Molka *et al.* [7] propose a set of benchmarks, including to study the performance details of the **Nehalem** architecture. Based on these benchmarks, they obtain undocumented performance data and architectural properties. This is the first work to measure the core-to-core communication overhead, but it is only applicable to the x86 architectures. Fang *et al.* extend the microkernels to Intel Xeon Phi [3]. Ramos *et al.* [9] propose a state-based modelling approach

for memory communication, allowing algorithm designers to abstract away from the architecture and the detailed cache coherency protocols. The model is built based on the measurement numbers of the cache-coherent memory hierarchy.

7 Conclusion

A variety of cache organizations and coherency protocols make modern multi-cores complicated, diverse, but hard-to-use. As the cache-based memory system is a critical factor that affects the overall performance, it is important to know its working mechanism and the achieved performance. This article focuses on dissecting the memory hierarchy of the Phytium 2000+ architecture with microbenchmarks. Specifically, we quantify the on-core and core-to-core communication performance when cachelines are in different states and located in various cache levels. We choose Phytium 2000+ as our experimental platform to access the performance of its memory system and dissect its working mechanism. The experimental results provide a detailed and quantitative performance description of the Phytium 2000+ memory hierarchy. We also compare architectural properties between Phytium 2000+ and the x86 architecture. For future work, we will use the hardware counters in our micro-benchmarks to collect detailed performance data, aiming to obtain more details of the memory system, e.g., on the TLB miss rate.

Acknowledgment. This work was funded by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, the National Natural Science Foundation of China under Grant agreements No. 61972408 and 61602501.

References

1. ARM, A.: NEON programmer's guide (2013)
2. Babka, V., Tůma, P.: Investigating cache parameters of x86 family processors. In: Kaeli, D., Sachs, K. (eds.) SBW 2009. LNCS, vol. 5419, pp. 77–96. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-93799-9_5
3. Fang, J., Sips, H.J., Zhang, L., Xu, C., Che, Y., Varbanescu, A.L.: Test-driving intel Xeon Phi. In: Lange, K., Murphy, J., Binder, W., Merseguer, J. (eds.) ACM/SPEC International Conference on Performance Engineering. (ICPE 2014), Dublin, Ireland, 22–26 March 2014, pp. 137–148. ACM (2014). <https://doi.org/10.1145/2568088.2576799>
4. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. *IEEE Comput.* **41**(7), 33–38 (2008). <https://doi.org/10.1109/MC.2008.209>
5. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. *IEEE Comput. Soc. Tech. Committee Comput. Archit. (TCCA) Newsl.* **2**(19–25) (1995)
6. McVoy, L.W., Staelin, C.: lmbench: portable tools for performance analysis. In: Proceedings of the USENIX Annual Technical Conference, 22–26 January 1996, San Diego, California, USA, pp. 279–294. USENIX Association (1996)

7. Molka, D., Hackenberg, D., Schöne, R., Müller, M.S.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12–16 September 2009, Raleigh, North Carolina, USA, pp. 261–270. IEEE Computer Society (2009). <https://doi.org/10.1109/PACT.2009.22>
8. Phytium: Mars II - microarchitectures. https://en.wikichip.org/wiki/phytium/microarchitectures/mars_ii
9. Ramos, S., Hoefler, T.: Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing, pp. 97–108 (2013)
10. Rutland, M.: Stale data, or how we (mis-) manage modern caches (2016)
11. Zhang, C.: Mars: a 64-core ARMv8 processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS), pp. 1–23. IEEE (2015)



TSU: A Two-Stage Update Approach for Persistent Skiplist

Shucheng Wang and Qiang Cao[✉]

Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology, School of Computer Science and Technology, Huazhong University of Science and Technology, Ministry of Education of China, Wuhan, China
{wsczq, caoqiang}@hust.edu.cn

Abstract. Skiplist, a widely used in-memory index structure, could incur crash inconsistency when running on emerging NVRAM (Non-Volatile Random Access Memory). Logging or strict serialization can ensure crash consistency at the cost of severe performance degradation. In this paper, we propose *TSU*, a *Two-stage* update approach to improve the performance of persistent skiplist while preserve crash consistency. TSU exploits space locality of skiplist and atomic write of NVRAM, thus effectively reducing expensive *cache line flush (clflush)* operations. To this end, we category all four crash inconsistent states into two types: recoverable and unrecoverable. TSU could guarantee the crash state is recoverable by constraining the memory access order for insertion and deletion. We further design a persistency algorithm to reduce clflush by preserving the memory persistent order of skiplist update. In addition, we develop a concurrent search for TSU. The evaluation result shows that TSU can reduce cache line flush with up to 47.6%, and decrease the average request latency by up to 36% for insertions compared to the strict serialization.

Keywords: Skiplist · NVRAM · Crash consistency

1 Introduction

Skiplist is a popular in-memory index structure and prevalently employed in key-value (KV) stores approaches [6, 7, 13, 21] to speed up query. Meanwhile, emerging non-volatile random access memory (NVRAM), such as PCM [26], STT-RAM [9] and 3D-Xpoint [15] presents DRAM-like read latency but could store data persistently. Therefore, a persistent skiplist running on NVRAM is desired to keep data even after a failure.

However, conventional skiplist directly deployed in NVRAM cannot ensure crash consistency. A skiplist update operation involves multiple memory accesses, which could be partly complete on crashing, leading to inconsistency in NVRAM. Traditionally, write-ahead-logging (WAL) and strict serialization could ensure

crash consistency in memory, and have been used to realize persistent B+-tree [27] and B-Tree [14]. Unfortunately, these two mechanisms are expensive. WAL writes all the updates twice. The strict serialization needs plenty of memory barriers and cache line flushes.

To address the crash inconsistency problem for skiplist, we propose *TSU*, a *Two-Stage* update approach to reduce the *cache line flush (clflush)* operations of skiplist update while ensuring crash consistency. The key idea of TSU is to exploit space locality of skiplist by effectively leveraging atomic write of NVRAM. We first reveal there are four crash inconsistent states for a skiplist. These four states can be divided into two types: recoverable and unrecoverable. Second, we design Two-stage update to ensure the crash states of skiplist are recoverable.

The rest of the paper is organized as follows: in Sect. 2, we present the background and challenges in designing a persistent skiplist. In Sect. 3, we propose the implementation of Two-stage Update. Section 4 evaluates the performance of TSU and Sect. 5 discusses related work. In Sect. 6, we conclude this paper.

2 Background and Challenge

2.1 Skiplist

Skiplist is originally introduced by Pugh et al. [17]. It has been extensively adopted to current prevalent key-value (KV) stores, such as HBase [6], LevelDB [7], MemSQL [21], because it is simple to build bottom-up and it maintains highly stabilizable index structure without complex rebalancing like B-tree in runtime.

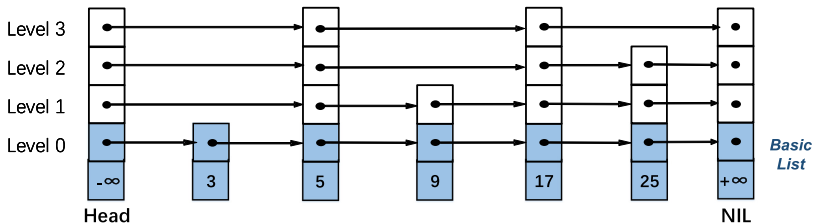


Fig. 1. Example of a skiplist with 4 levels

In a skiplist as shown in Fig. 1, a node contains a *key* that uniquely represents this node and *internal pointers* that point to its successor nodes. For a node, there exist *external pointers* in its forward nodes pointing to it. Each node has a level count, which is the number of internal pointers the node contains [17]. The level i internal pointer points to the successor node at level i . The level of a node is generated randomly by the probability factor k . For example, $k = 4$ means the level of a node will be 2 in the probability of 25%, and be 3 in the probability of 12.5%.

In a vertical view, a skiplist is comprised of multiple levels of linked lists. The bottom level is an ordinary ordered linked list containing real KV data. We define it as *basic list* for it is the cornerstone of skiplist. All the internal and external pointers except the basic list are *index pointers*. When inserting or deleting a node, a skiplist has to update both the basic list and all index pointers in a transactional way.

2.2 NVRAM

Emerging NVRAM (Non-Volatile Random Access Memory) such as PCM [26], STT-MRAM [9] and 3D Xpoint [15] are byte-addressable and non-volatile. The read access latency of NVRAM is comparative to DRAM, but the write latency could be 5–10x slower [16]. When compared with SSD (Solid State Disk), NVRAM is expected to provide 10-100x lower read and write latency [12]. Additionally, NVRAM commonly provides memory interfaces such as load and store, instead of block interface for storage.

2.3 Challenge

The challenge in designing a persistent skiplist is that it is hard to guarantee crash consistency. As mentioned in Sect. 2.1, the skiplist update demands a transaction containing multiple memory accesses. Without keeping transactional updates to NVRAM, the skiplist can partially update and persist under power failure, thus leading to an inconsistent state in NVRAM.

Crash Inconsistency States. Crash inconsistency refers to the inconsistent state caused by partial and disordered persisted data after a system crash. Three main reasons for the inconsistency are listed as follows: (1) The skiplist update operations contain multiple memory accesses. (2) Modern processors reorder

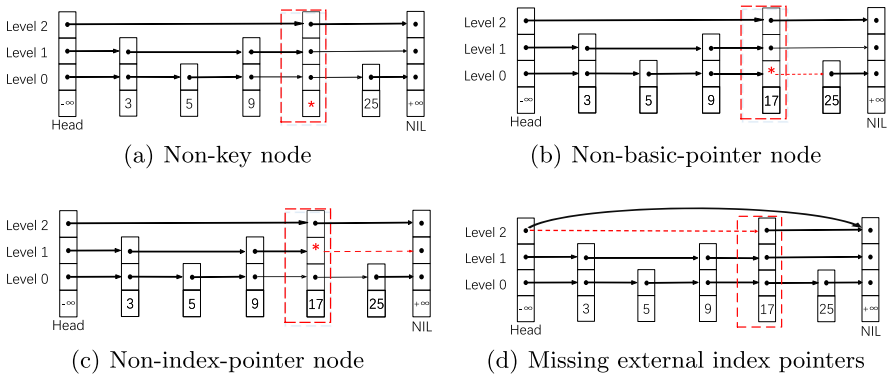


Fig. 2. Four crash inconsistent states of a conventional skiplist during inserting key 17.

memory operations [10]. (3) The CPU Cache is still volatile and all memory writes that are not persisted in NVRAM will be lost.

For instance, if we insert key 17 to a conventional skiplist without ordering memory writes, any update for key, internal or external pointers in skiplist could be lost after a system failure. It causes four inconsistent states as shown in Fig. 2(a)–2(d).

The Non-key Node in Fig. 2(a) is the node without a valid key. The Non-basic-pointer Node is the node with an uninitialized internal pointer in basic list, which could lose previously existing nodes like node 25 in Fig. 2(b). Figure 2(c) shows a Non-index-pointer node with uninitialized internal index pointer, which could point to an illegal address and cause fatal memory leak. All of them cause wrong result of search or update operations, and need to be repaired immediately after crash. Thus we categorize them into the *unrecoverable* type. Besides, missing external pointer like Fig. 2(d) is *recoverable* type of inconsistent state. It violates the basic semantics of the skiplist, but just affects search performance.

Therefore, when updating skiplist, we should carefully constrain the memory write order to avoid crash inconsistency. If all the unrecoverable states are avoided, the skiplist could be used immediately after a crash without reconstruction. We explicitly make use of memory fence and cache line flush instructions (*mfence* and *clflush* in Intel x86 architecture) [3, 10, 11, 27].

Granularity Mismatch. NVRAM could replace conventional DRAM placed on the memory bus, which makes a cache line (64 bytes in Intel x86 architecture) as a basic unit switching between cache and memory. We could use Intel’s Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) to support atomic cache line writes to NVRAM [18]. However, the demands for failure atomicity in NVRAM is at a smaller granularity (8 bytes). When updating skiplist with strict serialization by performing an atomic write to NVRAM, we have to flush a cache line for each memory write. This scheme could lead to a 64-bytes cache eviction for merely an 8-bytes update, which could cause low cache hit rate. To alleviate such cost, we should flush a cache line containing multiple dirty data. Besides, even the strict serialization model could not atomically flush multiple cache lines [20]. If a system crashes while flushing multiple cache lines, the consistency could not be guaranteed. Therefore, it is necessary to constrain the cache flush order as long as the skiplist node size is larger than a single cache line.

3 Two-Stage Update

3.1 Design of TSU

To guarantee the crash state is always recoverable, we divide the insertion operation into two consecutive stages as shown in Fig. 3: the *Node modification stage* updates the basic list and internal pointers in the target node. The *Index modification stage* updates other external pointers in its involving forward nodes.

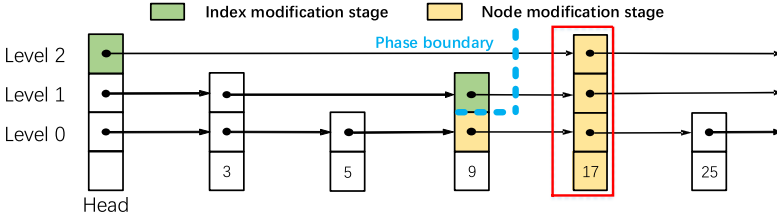


Fig. 3. Two-stage update when inserting node 17. The insertion consists of two stages: the node modification phase and the index modification stage.

These two stages serialize globally: one stage must complete before the other so all dirty cache lines produced in the previous stage could persist in NVRAM.

The node modification stage is a KV persistent process that ensures KV data to be both recoverable and visible. For insertion, it allocates the new node but doesn't initialize all elements immediately. In order to keep recoverable crash states, the basic list should modify first before others. Then the new list node in the basic list should persist with cflush. At that time, the node with durable KV is visible for search operations even if it is incomplete, which doesn't affect concurrent search. This is because the search threads could visit the new node just through the basic list in the bottom level, and other uninitialized pointers will not be used. At the end of this stage, we update and persist other internal pointers in high levels. Then, the index modification stage updates all the external pointers except the one in the basic list. For insertion, it requires a specific bottom-up order to modify external pointers to preserve consistency while leveraging the external pointer locality. For deletion, the process is reversed. We will describe the detail in the next section.

3.2 The Persistency Algorithm in TSU

In both node and index modification stage, we need to determine the update order of pointers to avoid inconsistent state. One intuitive way is strict serialization with cflush for each memory write, which is expensive. In Fig. 4, we evaluate the cflush counts produced in 100k random insertion with 8-bytes key under strict serialization. We change the average level of skiplist by adjusting the factor k as mentioned in Sect. 2.1. We observe that with a different value of k , each insertion needs 5.2 to 7 times of cflush operations.

In order to reduce the number of cflush operations, we design a persistency algorithm for TSU to combine multiple memory writes into a cache line when updating index pointers. We observe that some of the updating pointers are continuous in memory address and exhibit obvious space locality, and there is no semantic dependency between them. It is widely common when updating a skiplist. If we do not explicitly perform cflush, the CPU could evict dirty cache lines randomly [10]. It means CPU could reorder these pointer updates into multiple physical memory flushes and break the potential locality. Therefore,

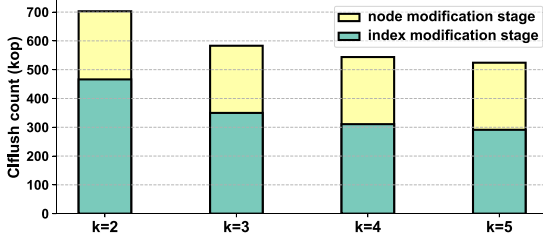


Fig. 4. The clflush counts produced by 100k random insertion with different probability factor k , which is counted in index modification stage and node modification stage respectively.

to leverage the locality of pointers, we design an algorithm to insert a copy operation between consecutive memory writes. The copy operation reads the value of first write, and then copies it to the destination of the following write, which will update with a new value finally. The extra copy explicitly produces Write-After-Write (WAW) and Read-After-Write (RAW) dependency between the two updates. CPU with TSO does not reorder WAW and RAW at the same memory address [19]. As a result, if one pointer updates successfully, its preceding pointers must have been flushed [25]. Besides, the copy leads to a read-hit and a write-hit in cache, avoiding extra actual memory access. With this approach, TSU can persist all of them simultaneously with an atomic clflush operation.

If the node size spans across multiple cache lines, TSU writes a cache line and flushes it with `mfence` before writing next cache line. This approach ensures the flush order is not to change.

Detectable Duplicate Pointer. The extra copy approach results in *duplicate pointers*, which means that its adjacent pointer updates incorrectly. This inconsistency is recoverable but needs to be identified. However, conventional skiplists allow pointers with same value in a node. To avoid the *duplicate pointers* in traditional practices, we define the internal pointers merely point to its successor pointer at the same level instead of the successor node. Therefore, a pointer in each node has different value in normal cases, thus removing *duplicate pointers* in the general process. The duplication case uniquely represents a transient inconsistency that can be easily detected. Please note the method could lead to indirect read access to a node. A macro (e.g., the `container_of()` in Linux) is used to get the address of the node by calculating the memory offset of its internal pointer.

Insertion. The insertion operation of TSU shown in Fig. 5 is an example of inserting a KV pair (17, value) into skiplist. A similar procedure could apply to deletion.

First, once the new node allocates, the corresponding node modification stage begins. TSU initializes the new KV and internal pointer in the basic list

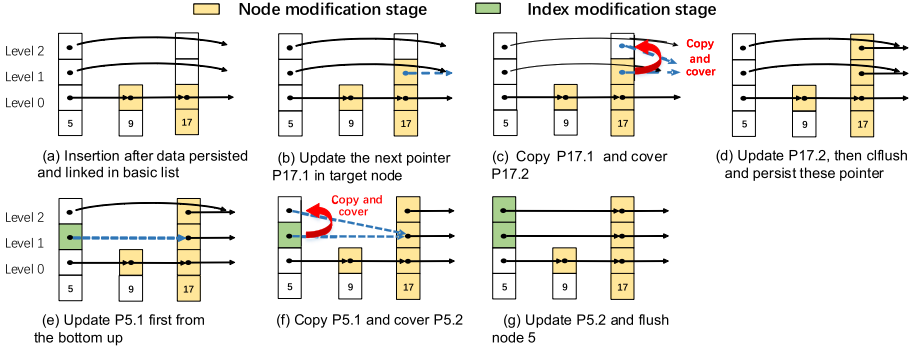


Fig. 5. The persistency algorithm in TSU when inserting the node 17. The insertion first updates and persists the basic list, then updates other internal and external pointers with the persistency algorithm.

(assuming their total size less than a cache line) with *clflush + mfence*. After that, TSU atomically updates forward pointer P9.0. At this point, the node has been inserted in the basic list as Fig. 5(a) shows.

Next, TSU updates the node’s other internal pointers at a high level. There are three steps explained in Fig. 5(b)–5(d). First, TSU initializes the pointer P17.1. Since other internal pointers in the same node also need to update, TSU does not perform cflush immediately. Second, before updating the next pointer P17.2, TSU copies P17.1 and overwrites P17.2 like Fig. 5(c). The copy produces a RAW to P17.1, and creates a memory write to P17.2 at the same time. Third, TSU updates P17.2, which causes a WAW similarly. This process builds a dependency constraint on the execution order of memory write based on TSO. Finally, in Fig. 5(d), there is no uninitialized internal pointer in node 17, TSU atomically flushes cache line with multiple dirty pointers in it. Similarly, TSU updates external pointers (P5.1, P5.2) in the index modification stage as shown in Fig. 5(e)–(g) to realize crash consistency.

Suppose crash occurs at the state in Fig. 5(c) or 5(f). Although the duplicate pointers can be detected in NVRAM, they violates our rule. At this point, one of the duplicate pointers is correct, and the inconsistent state can recover by detecting and ignoring the wrong pointer (P17.2 and P5.2 in this example). In addition, if crash occurs at the state in Fig. 5(b), it is a recoverable state. This is because the uninitialized internal pointers could not be accessed by previous index at the same level. If a crash happens when the index modification stage is like Fig. 5(e) shows, it will result in a recoverable and tolerable state as shown in Fig. 2(d).

Recovery. To perform recovery after a crash, we should first find the entrance of skiplist in NVRAM, then repair the inconsistency in it. The entrance of skiplist can be defined as a specific address in NVRAM. This function is supported by previous work such as Makalu [1].

Then, the recovery procedure traverses the skiplist to find the inconsistent states. With TSU, the recovery could just check two kinds of recoverable inconsistent states in the skiplist after power failure:

1. There are duplicate pointers in a node.
2. There are missing index pointers at top-level as Fig. 2(d) shows.

For state 1, one of the duplicate pointers is pointing to a successor pointer at different level, which is a *illegal* pointer. To repair it, the recovery modifies the *illegal* pointer by pointing it to its successor pointer at the same level. For instance, if the duplicate pointers are generated by insertion, the *illegal* pointer is the one at a higher level, and it is reverse for deletion. For state 2, the recovery repairs all the missing pointers by pointing them to their successor pointers at the same level.

3.3 Concurrent Search

In TSU, the persistency algorithm could lead to duplicate pointers when inserting a node, which could affect the correctness of original concurrent search manner. An intuitive solution is to design a lock-based skiplist to maintain concurrency between search and insertion. However, lock is a heavyweight operation that limits the scalability of concurrency. To implement the concurrent search for TSU without lock, search operation should detect and tolerate duplicate pointers to ensure correct result when it traverses to the node in updating. As mentioned above in Sect. 3.2, one of the duplicate pointers is *illegal* that could lead to wrong search results. If searches detect the illegal pointer, it could continue searching by the other one. Search threads could detect the *illegal* pointer by determining whether it has the same level as its successor pointer. This process performs an atomic memory reads and doesn't need an extra lock.

4 Evaluation

We run experiments on a server that has two Intel Xeon E5-2696 v4 processors (2.20 GHz, 22 CPUs) and 64 GB DRAM. We use a DRAM-based NVRAM latency emulator-Quartz [23] that is widely used in previous studies [10, 24]. We compare three variants of persistent skiplist. The **DRAM-SS** is a conventional skiplist that uses *clflush* and *mfence* for each memory write to keep crash consistency. The **TSU-SS** issues a Two-stage update and strict serialization in the index modification stage. **TSU-Atomic** further updates multiple external pointers in the index modification stage with the persistency algorithm.

4.1 Performance

We warm up with 500k 8-bytes random integer keys, then execute 100k random insertions and measure the average request latency (i.e., the process time of an operation) and *clflush* counts. The skiplist nodes is aligned with cache line

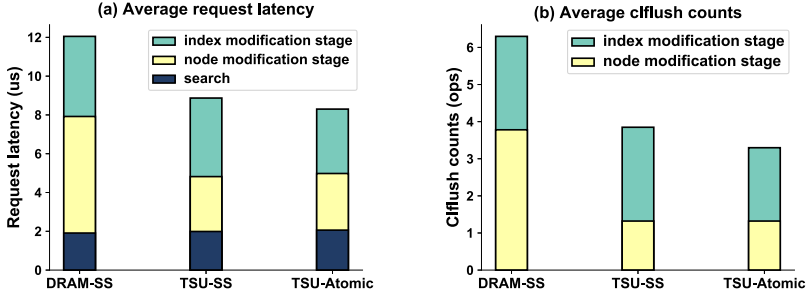


Fig. 6. The average request latency and cflush counts of random insertions with three different kinds of persistent skiplist. The latency of insertion is divided into three parts: the index modification stage, the node modification stage and search.

and the maximum node size is set to 4 cache lines (256 bytes). The NVRAM read/write access delay is set to 600 ns.

Figure 6 shows that TSU-Atomic and TSU-SS reduce the request latency compared with DRAM-SS in the node modification stage and the index modification stage, respectively. Figure 6(a) shows that TSU-SS reduces 52.9% latency of the node modification stage compared with DRAM-SS and 26.3% of the insert operation. This is because TSU-SS uses the persistency algorithm for TSU, which reduces 65.8% cflushes in the node modification stage as shown in Fig. 6(b). Moreover, TSU-Atomic effectively reduces more than 18% latency of the index modification stage compared with TSU-SS. In general, TSU-Atomic decreases the total insertion latency by 30.6% compared with DRAM-SS.

4.2 NVRAM Latency Effect

We demonstrate how NVRAM write latency affects the insert performance as shown in Fig. 7. We set NVRAM read access delay to 300 ns while increasing only the write access delay from 300 ns to 1200 ns, and measure the average latency

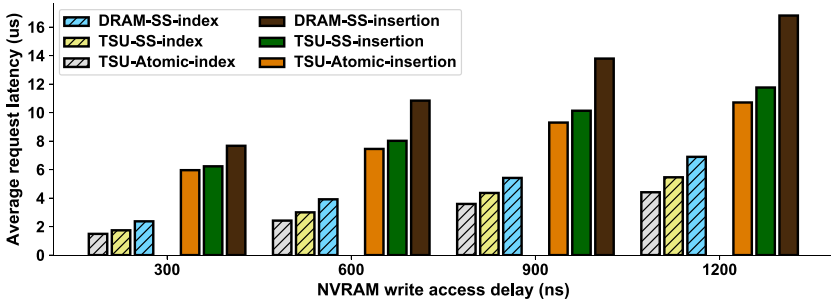


Fig. 7. Different read-write delay ratio could effect the average latency for each index modification stage and insertion.

for each insertion operation (with the **-insertion** suffix) and index modification stage (with the **-index** suffix) in different skiplist.

When we change the write latency of NVRAM, the insertion time decreases from 22% to 36% in TSU-Atomic compared with DRAM-SS, and the average latency of the index modification stage decreases from 11.9% to 19.5%. This is because the main cost in both node and index modification stages is cache line flush and memory write, which is more sensitive to memory write latency. Therefore, the persistent skiplist with TSU is more applicable to NVRAM with asymmetric latency, such as PCM.

4.3 Concurrency

In the experiment described in Fig. 8, we evaluate the throughput of three concurrent persistent skiplists. **LOCK** is a basic concurrent version of DRAM-SS, which requires read-write lock. **TSU-SS** and **TSU-Atomic** could support concurrent searches without locks, although they still need write locks to serialize insertions. The search operation of TSU-SS does not need to tolerate duplicate pointers while the TSU-Atomic has to, for TSU-SS does not cause duplicate pointers in the index modification phase. Our write lock uses the `STD::mutex` class in `c++ 11` and read-write lock uses the `boost::shared_mutex` class. We compile the TSU program set with `-O0` optimization option here because the compiler optimization could reorder instructions and affect the correctness of lock-free concurrent search [10]. In this experiment, the NVRAM latency is set to 600 ns. We insert 500k 8-bytes random keys to warm up, and then launch multiple searches and a single insertion with 8:2 search and insert load.

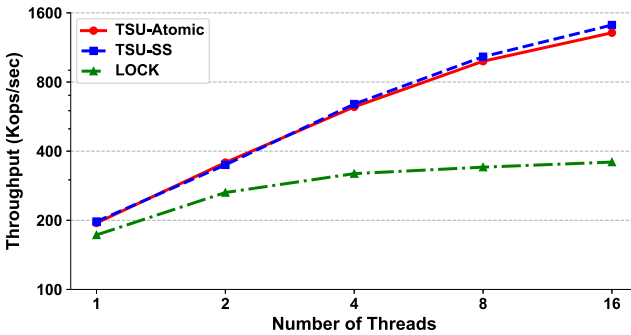


Fig. 8. The throughput performance with varying number of concurrent threads.

Result in Fig. 8 shows that TSU-Atomic and TSU-SS could benefit from concurrent search, and their throughput performs 7.1x and 6.8x speed-up when the number of search threads increases from 1 to 16, respectively. TSU-Atomic decreases the throughput slightly but has no affect on the concurrency extendibility. This is because TSU-Atomic modifies the search operation and causes a little

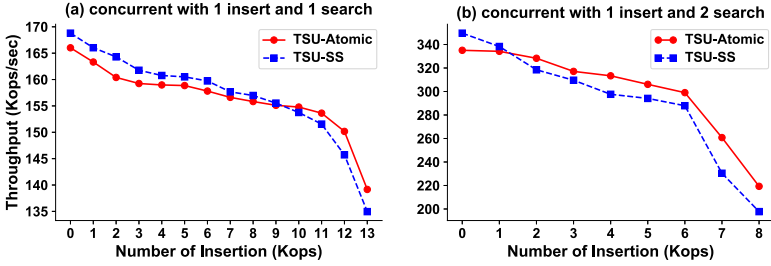


Fig. 9. Concurrent performance of TSU-Atomic and TSU-SS with 300k search load and varying numbers of insertion load.

overhead (i.e., less than 0.4% throughput reduction for a single thread) as mentioned in the previous Sect. 3.3. For LOCK, the speed-up becomes saturated at 4 search threads, because read-write locks do not allow concurrency between read and write, so the throughput is limited by insertion.

Figure 9 presents that TSU-Atomic could reduce the adverse impact caused by insertion under read-write concurrency. We compare the performance of TSU-Atomic and TSU-SS with single insertion thread and multiple search threads. We use the same settings as shown above, then launch 300k random search with varying numbers of insertions respectively at the same time. As the insertion operation inserts more nodes, the overall throughput decreased for the search time extends. TSU-Atomic could tolerate these declines more than TSU-SS because it shortens the insertion time in the index modification stage, which allows concurrent search to utilize high-level index pointers earlier. In addition, when we insert more than 120,000 (60,000 with two search threads in Fig. 9(b)) keys, the throughput decreases significantly because the overall execution time of insertion exceeds search at this point.

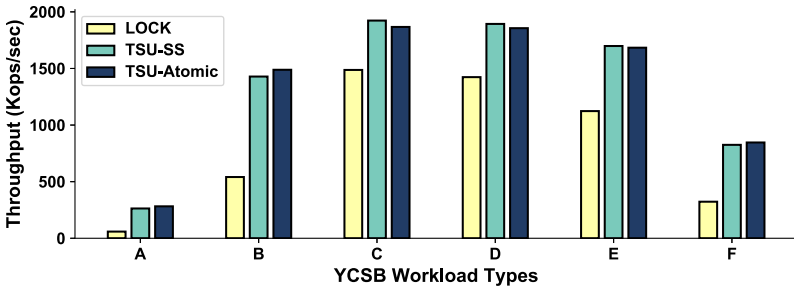


Fig. 10. Overall throughput of persistent skiplists with YCSB workloads.

4.4 YCSB

To evaluate the performance of TSU under real world scenarios, we use the YCSB [4] benchmark and run the following workloads: A (update heavy: 50/50), B (read mostly: 95/5), C (read only), D (read recently), E (range query) and F (read-modify-write). We first launch a warm-up with 500k insertions and then perform 100k records. We use YCSB version 0.12.0 to generate workloads and replay them with 16 threads.

Figure 10 presents the throughput of three different kinds of persistent skiplists. TSU is more effective with larger proportion of insertions. It improves throughput performance up to 4.6x in workload A, 2.6x in workload B and 2.5x in workload F compared with LOCK, respectively. Among the read-only workloads C, D and E, the max throughput of TSU is 1.2x larger than LOCK. This is because the read-write lock in LOCK has extra memory access cost than the concurrent TSU. In addition, TSU-SS and TSU-Atomic have little difference of performance in all five YCSB workloads. TSU-Atomic improves 6.5% throughput in the update heavy workload A but decreases 3.5% under the read only workload C compared with TSU-SS. This is because TSU-Atomic performs less cflashes in insertion, while having a constant overhead in searching.

5 Related Work

Index Structures for Block Devices. Several studies attempt to redesign persistent B-tree and B+-tree for NVRAM [3, 10, 14, 16, 22]. There are different challenges between persistent B-tree and skiplist. Inserting a new KV entry into a B-tree node will shift half of the existed entries on average, while skiplist does not move any existed elements. To maintain crash consistency, B-tree or B+-tree has to perform multiple cflflush and mfence operations for each entry shift. The wB+-Tree [3] designs unsorted nodes to avoid shifting entries, and it introduces a bitmap as metadata in each node to make search efficient. However, the cflflush produced by wB+-Tree is too much mainly because the metadata brings extra persistent demands. FAST&FAIR [10] intends to solve granularity mismatch problem in B+-Trees. It keeps entries in sorted order without setting metadata like the wB+-Tree, and it could persist multiple entry shift operations with a single cflflush to keep crash consistency. This method inspires our work. DPTree [28] designs a two-level persistent index architecture for B+-Tree in DRAM-NVRAM hybrid systems. It batches multiple writes in DRAM and later merges them into NVRAM to reduce persistence overhead.

Index Structures in Memory. Some studies [11, 12] design persistent skiplists of Log-Structure Merge Trees (LSM-Trees) to deliver low search latency and high throughput. They set no consistency mechanism for the index pointers of skiplist because they could be reconstructed upon system crash. Similarly, NV-skiplist [2] designs a persistent skiplist with separated data and indexes. It sets up the basic list in NVRAM for data persistence and builds index pointers in DRAM to

retain performance. However, all index pointers will be lost after system crash, which could be expensive to recover in a large-scale skiplist.

Concurrent Skiplist. Skiplist can support concurrent workload [21] to enhance throughput. Related works [5, 8] utilize the *Compare_And_Swap* (CAS) synchronization primitive to provide lock-free skiplist for high concurrency. However, it is too complex to make it satisfy crash consistency. Therefore, we design a simple implementation with concurrent read and single write, which is not totally lock-free but could be adequate to our environment.

6 Conclusion

In this work, we present TSU, a *Two-stage* update approach for persistent skiplist. TSU keeps crash consistency without logging or strict serialization. It constrains the update order of skiplist, thus guaranteeing that the crash inconsistent state is recoverable. By leveraging the space locality of the skiplist and memory order constrains following Total Store Ordering, we design a persistency algorithm for TSU that could perform failure atomic skiplist update. The algorithm ensures that multiple dirty pointers in skiplist can atomically write back to NVRAM. Besides, we enable a concurrent search without locks.

Acknowledgment. This work is supported in part by National key research and development program of China under Grant 2018YFA0701804 and Grant 2018YFA0701805, in part by the Creative Research Group Project of NSFC No. 61821003.

References

1. Bhandari, K., Chakrabarti, D.R., Boehm, H.J.: Makalu: fast recoverable allocation of non-volatile memory. In: ACM SIGPLAN Notices, vol. 51, pp. 677–694. ACM (2016)
2. Chen, Q., Yeom, H.: Design of skiplist based key-value store on non-volatile memory. In: 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), pp. 44–50. IEEE (2018)
3. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. Proc. VLDB Endow. **8**(7), 786–797 (2015)
4. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
5. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, pp. 50–59. ACM (2004)
6. George, L.: HBase: The Definitive Guide: Random Access to Your Planet-Size Data. O’Reilly Media, Inc., Sebastopol (2011)
7. Ghemawat, S., Dean, J.: LevelDB (2011)

8. Herlihy, M.: A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **15**(5), 745–770 (1993)
9. Huai, Y., et al.: Spin-transfer torque MRAM (STT-MRAM): challenges and prospects. *AAPPS Bull.* **18**(6), 33–40 (2008)
10. Hwang, D., Kim, W.H., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent B+-tree. In: 16th USENIX Conference on File and Storage Technologies. (FAST 2018), Oakland, CA, pp. 187–200. USENIX Association (2018)
11. Kaiyakhmet, O., Lee, S., Nam, B., Noh, S.H., Choi, Y.: SLM-DB: single-level key-value store with persistent memory. In: 17th USENIX Conference on File and Storage Technologies, (FAST 2019), Boston, MA, pp. 191–205. USENIX Association (2019)
12. Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Redesigning LSMS for nonvolatile memory with NoveLSM. In: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 2018), pp. 993–1005 (2018)
13. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **44**(2), 35–40 (2010)
14. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: FPTree: a hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In: International Conference on Management of Data (2016)
15. Packard, H.: Understanding the Intel/Micron 3D Xpoint Memory (2015)
16. Ping, C., Lee, W.C., Yuan, X.: Making B+-tree efficient in PCM-based main memory (2014)
17. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* **33**(6), 668–677 (1990)
18. Rao, D.S., et al.: System software for persistent memory. In: Bulterman, D.C.A., Bos, H., Rowstron, A.I.T., Druschel, P. (eds.) Ninth Eurosys Conference 2014, (EuroSys 2014), Amsterdam, The Netherlands, 13–16 April 2014, pp. 15:1–15:15. ACM (2014)
19. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D.: Memory access scheduling. *ACM SIGARCH Comput. Archit. News* **28**(2), 128–138 (2000)
20. Seo, J., Kim, W., Baek, W., Nam, B., Noh, S.H.: Failure-atomic slotted paging for persistent memory. In: Chen, Y., Temam, O., Carter, J. (eds.) Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2017), Xi’an, China, 8–12 April 2017, pp. 91–104. ACM (2017)
21. Shamgunov, N.: The MemSQL in-memory database system. In: IMDM VLDB (2014)
22. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R.H.: Consistent and durable data structures for non-volatile byte-addressable memory. In: USENIX Conference on File and Storage Technologies (2010)
23. Volos, H., Magalhaes, G., Cherkasova, L., Li, J.: Quartz: a lightweight performance emulator for persistent memory software. In: Proceedings of the 16th Annual Middleware Conference, pp. 37–49. ACM (2015)
24. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: ACM SIGARCH Computer Architecture News, vol. 39, pp. 91–104. ACM (2011)
25. Wang, Y., et al.: Robustness in the salus scalable block store. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, (NSDI 2013), Lombard, IL, USA, 2–5 April 2013, pp. 357–370. USENIX Association (2013)

26. Wong, H.S.P., et al.: Phase change memory. *Proc. IEEE* **98**(12), 2201–2227 (2010)
27. Yang, J., Wei, Q., Cheng, C., Wang, C., Leong, K., He, B.: NV-tree: reducing consistency cost for NVM-based single level systems. In: *USENIX Conference on File and Storage Technologies* (2015)
28. Zhou, X., Shou, L., Chen, K., Hu, W., Chen, G.: DPtree: differential indexing for persistent memory. *Proc. VLDB Endow.* **13**(4), 421–434 (2019)



NV-BSP: A Burst I/O Storage Pool Based on NVMe SSDs

Qiong Li¹(✉), Dengping Wei¹, Wenqiang Gao², and Xuchao Xie¹

¹ College of Computer, National University of Defense Technology, Changsha, China
qiong_joan_li@aliyun.com

² Beijing Memblaze Technology Co., Ltd., Beijing, China

Abstract. The High-Performance Computing (HPC) systems built for future exascale computing, big data analytics, and artificial intelligence applications raise an ever-increasing demand for high-performance and highly reliable storage systems. In recent years, as Non-Volatile Memory express (NVMe) Solid-State Drives (SSDs) are deployed in HPC storage systems, the performance penalty paid for the legacy I/O software stack and storage network architecture turns out to be non-trivial. In this paper, we propose NV-BSP, an NVMe SSD-based Burst I/O Storage Pool, to leverage the performance benefits of NVMe SSD, NVMe over Fabrics (NVMeoF) Protocol, and Remote Direct Memory Access (RDMA) networks in HPC storage systems. NV-BSP disaggregates NVMe SSDs from HPC compute nodes to enhance the scalability of HPC storage systems, employs fine-grained chunks rather than physical NVMe SSD devices as the RAID-based data protection areas, and exploits high concurrent I/O processing model to alleviate the performance overhead from lock contentions and context switches in critical I/O path. We implement NV-BSP in Linux and evaluate it with synthetic FIO benchmarks. Our experimental results show that NV-BSP achieves scalable system performance as the number of NVMe SSD and CPU core increases and obtains much better system performance compared with the built-in MD-RAID in Linux. Compared with node-local SSDs in HPC, NV-BSP provides a full system solution of storage disaggregation, delivers comparable performance, and significantly improves system reliability.

Keywords: Burst I/O Storage Pool · NVMe SSD · NVMe over Fabrics · High-Performance Computing

1 Introduction

High-Performance Computing (HPC) has proven its great power in facilitating data-driven scientific discovery [10]. Future HPC systems will be not only built for large-scale scientific computing, but also for big data analytics and artificial intelligence applications, which raises an ever-increasing demand for high-performance and highly reliable storage systems [13, 15].

Conventional Hard Disk Drive (HDD)-based RAID architectures [2] have been used as a key component of HPC storage systems over the past 30 years. However, due to the inherent mechanical characteristics of the rotating media in HDDs, HDD-based storage arrays are unable to meet the high IOPS, high bandwidth, and low latency requirements of the HPC applications with data-intensive problems [7, 20, 24]. The new emerging NAND Flash-based Solid-State Drive (SSD) provides orders of magnitude lower latency and consumes less power than HDDs [17–19]. SSDs are considered to entirely replace HDDs in future HPC storage systems. SSDs are first designed as the drop-in replacements of traditional hard disks with interfaces like SATA and SAS. However, the interface protocols were designed for hard disks which significantly limit the output performance of SSDs. This promotes the design and development of the Non-Volatile Memory express (NVMe) protocol that is capable of leveraging the internal parallelism of SSDs and reducing the software overhead in the I/O path [14, 16, 22, 23]. NVMe SSDs have been rapidly emerging on the storage market and will be widely deployed in both data center and HPC storage systems soon.

As HPC systems have an urgent need for high-performance and highly reliable large-scale storage systems, using NVMe SSDs to build All Flash Array (AFA) can effectively meet the requirements simultaneously. However, existing storage array architectures have critical limitations in terms of software overhead in I/O path and parallelism exploitation of NVMe SSDs [18, 19, 21, 23]. The workload characteristics of a certain HPC application can be latency-sensitive or throughput-oriented or continuously changing during the application lifespan. As the aggregate bandwidth can be easily achieved in large-scale parallel storage systems, obtaining low latency is more challenging. As the scalability of the PCI express (PCIe) bus cannot satisfy the connections of a large amount of NVMe SSDs in large-scale storage systems, NVMe over Fabrics (NVMeoF) protocol is proposed to extend the advantages of NVMe protocol to shared storage architecture [4, 6]. NVMeoF offers a solution that separates storage from HPC compute node (CN) and connects storage to CN through a network fabric. Currently, NVMeoF can adequately support fabric transports like Remote Direct Memory Access (RDMA), TCP, and Fibre Channel (FC), how to efficiently integrate NVMeoF with in-house interconnection networks of specific HPC systems remains stagnant.

In this work, we consolidate the storage array trend towards integrating NVMe SSDs and NVMeoF target in a single storage server. We propose NV-BSP, an NVMe SSD-based Burst I/O Storage Pool, to leverage the performance benefits of NVMe SSD, NVMeoF Protocol, and RDMA networks in HPC storage systems. Specifically, NV-BSP disaggregates NVMe SSDs from HPC compute nodes, which improves the storage resource utilization and enhances the scalability of HPC storage systems. NV-BSP employs fine-grained chunks rather than physical NVMe SSD devices as the RAID-based data protection areas, which avoids an entire NVMe SSD participating data reconstruction and achieves load balance without data redirection or migration. NV-BSP exploits a high concurrent I/O processing model to alleviate the performance overhead from lock

contention and context switch in critical I/O path, which enables the performance of NV-BSP increasing linearly with the number of NVMe SSDs and CPU cores.

We implement NV-BSP in Linux and evaluate it with synthetic FIO benchmarks. Our experimental results show that NV-BSP achieves scalable system performance as the number of NVMe SSD and CPU core increases and obtains better system performance compared with the build-in MD-RAID in Linux. Compared with node-local SSDs in HPC, NV-BSP provides a full system solution of storage disaggregation, delivers comparable performance, and significantly improves system reliability.

The rest of this paper is organized as follows. Section 2 provides an overview of NV-BSP, Sect. 3 describes the high concurrent I/O processing mechanism in NV-BSP. We evaluate the performance of NV-BSP in Sect. 4 and summarize the related work in Sect. 5. Finally, we conclude the paper in Sect. 6.

2 NV-BSP Overview

In this section, we give an overview of the system architecture, storage management, and storage disaggregation designs of NV-BSP.

2.1 System Architecture

The system architecture of NV-BSP is shown in Fig. 1. The hardware of NV-BSP includes the NVMe SSDs connected to CPU via PCIe bus, NVMeoF network interface for NVMeoF purpose, and other common components of storage servers, i.e., CPU, RAM, etc. The software of NV-BSP mainly composes of the storage resource manager, I/O processing handlers, and NVMeoF target. NV-BSP manages the data to underlying NVMe SSDs in the block layer in the I/O path. The storage resource manager is responsible for managing all the storage resources in NV-BSP and exporting virtual disks (VDisk) to applications, which will be discussed in detail in Sect. 2.2. I/O processing handlers will produce and activate independent threads to serve I/O requests, reconstruction requests, and error events, etc. NVMeoF target provides NVMe over RDMA communication between HPC compute nodes (CN) and VDIs in NV-BSP, which will be further discussed in Sect. 2.3.

2.2 Resource Management

Figure 2 describes the storage resource management mechanism in NV-BSP. The storage resource management mechanism can be divided into four levels include storage pool management, resource allocation, data protection, and VDisk management. In NV-BSP, NVMe SSDs are first organized as a storage pool, in which the logical address space of all the NVMe SSDs is divided into fixed-size chunks. The storage pool manager maintains the states of all the fine-grained chunks. Different from traditional RAID, NV-BSP completely separates physical resources

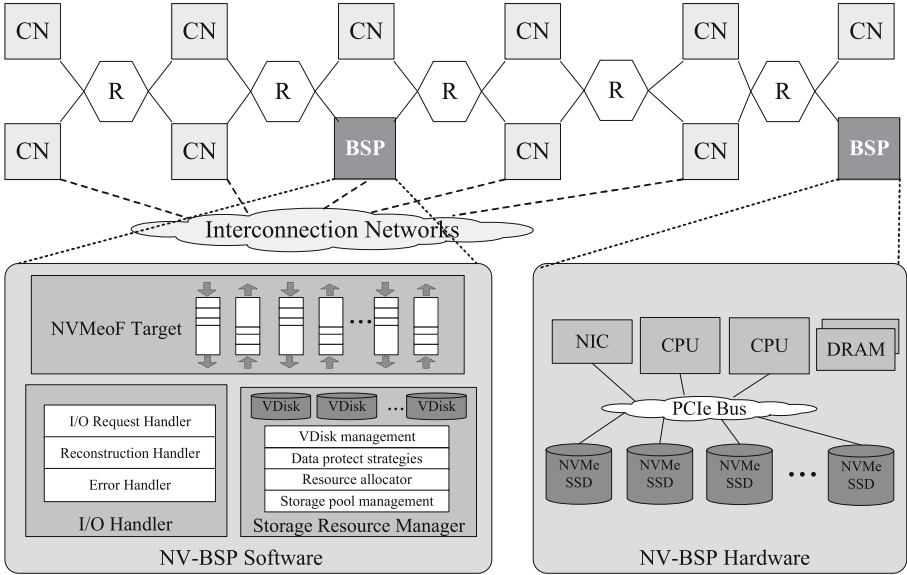


Fig. 1. NV-BSP system architecture.

and RAID-based data protection areas through chunk-level resource virtualization. Therefore, the data reconstruction operations in NV-BSP no longer relies on the entire SSD device to participate. Furthermore, through fine-grained division, the data written to the VDisks from the same NV-BSP will be evenly distributed on each NVMe SSD, which achieves load balancing among NVMe SSDs without data redirection and migration.

At the time of creating a VDisk from the storage pool, Resource allocator groups several chunks into a container which is divided into finer-stripes based on the configured data protection level (e.g., RAID 0/1/5/6). Stripe is the basic granularity of read and write operations in NV-BSP. VDisk is a collection of containers and will be exported as a logical volume in the operating system. The corresponding logical volume of the VDisk will provide storage services for upper-layer applications through a standard block device interface.

2.3 Storage Disaggregation

In NV-BSP, a single storage pool can export several VDisks simultaneously and the VDisks can be carved into NVMe namespaces with each namespace allocated to a specific HPC compute node. The I/O processing handlers are responsible for serving I/O requests concurrently. In NV-BSP, the NVMeoF target dynamically and arbitrarily attaches virtual disks with needed capacity and performance via QoS management technology directly to the compute node where the application runs on. As NVMeoF initiators, computing nodes send NVMeoF read/write commands through the RDMA network to the destination NV-BSP. The NVMeoF

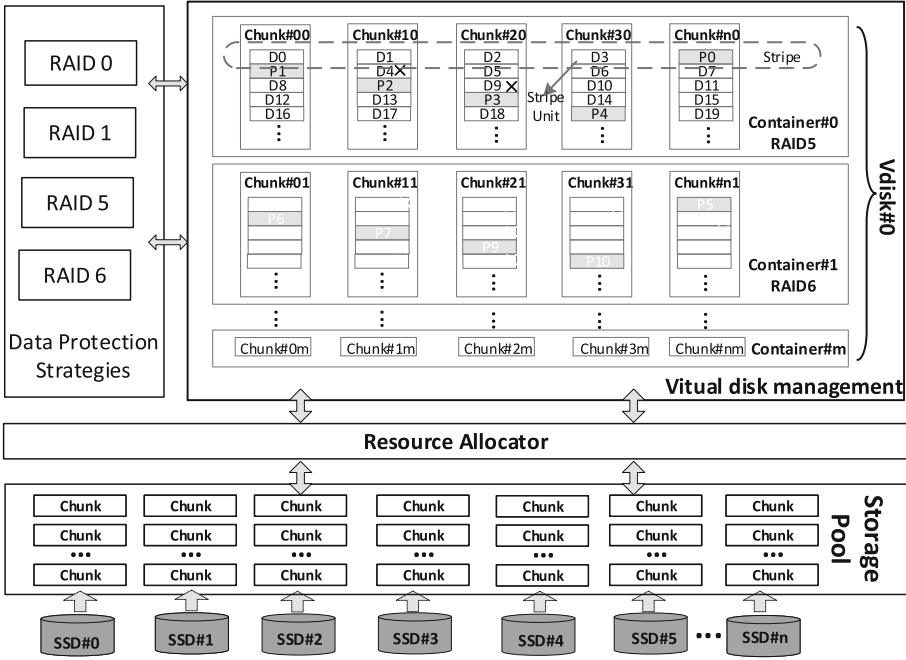


Fig. 2. Storage resource manager in NV-BSP.

target in NV-BSP parses the received NVMeoF commands and converts them into NVMe commands to a VDisk that exported from NV-BSP. The storage resource manager in NV-BSP manages all the VDisks, and finally transmits each NVMe command into multiple NVMe commands to the corresponding NVMe SSDs. Each NVMe SSD completes the subsequent read data from the NVMe SSD to the memory, or write data into the NVMe SSD, finally transmits the read/write completion message to NV-BSP. NV-BSP combines the arrived completion messages into a single NVMe response. Finally, the NVMeoF target in NV-BSP returns the NVMeoF response to corresponding compute nodes over the NVMeoF storage network.

3 High Concurrent I/O Processing

In this section, we describe the details of the high concurrent I/O processing model in NV-BSP.

3.1 Task Grouping

In NV-BSP, task grouping is designed to achieve high concurrent I/O processing. As NV-BSP serves as both NVMeoF target and RAID array simultaneously, significant CPU competitions between the two tasks will be introduced when

NVMeoF target and RAID array services are all enabled. In NV-BSP, CPU cores are divided into two different groups, i.e., NVMeoF Target Group (NT-Group) and RAID Array Group (RA-Group). The CPU cores in NT-Group will only be assigned for the NVMeoF target while that in RA-Group only assigned for the RAID array task. In this case, two different tasks will not run on the same CPU core anymore, which effectively alleviates CPU conflicts caused performance overhead and improves the concurrency of I/O processing.

In RA-Group, each CPU core occupies an independent data structure when performing an I/O handler. Thus, the CPU cores in RA-Group will no longer need to compete for a data structure and the lock contention overhead is eliminated. As shown in Fig. 3, for the I/O handlers that perform RAID tasks on N cores (indicated as CPU_1 to CPU_N), each I/O handler uses an independent data structure to avoid lock contentions among CPU cores, which enables the I/O performance increases linearly with the increase of the number of CPU cores in NV-BSP.

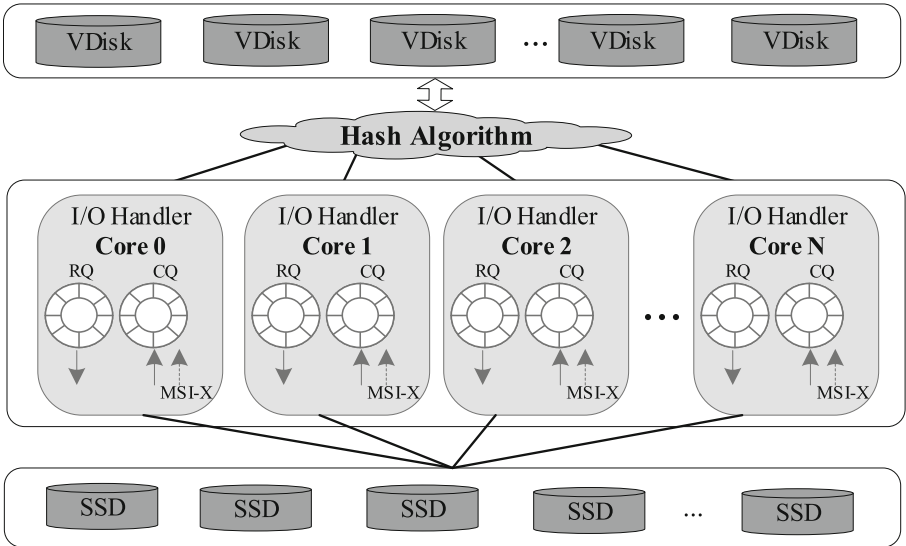


Fig. 3. High concurrent I/O processing model in NV-BSP.

3.2 I/O Handler Allocation

In NV-BSP, I/O handlers are responsible for handling I/O requests, including read and write requests from application and data reconstruction requests within NV-BSP. Each I/O handler thread processes I/O requests of different logical address areas in VDisk. The logical address space of a VDisk is divided into multiple regions that do not overlap with each other. I/O requests to a region are

processed by the corresponding I/O handler thread. As shown in Fig. 3, when an application accesses the VDisk in NV-BSP, several I/O handler threads will wake up according to the hash algorithm. Since there is no access correlation between the requests handled by different I/O handlers, these I/O handler threads can run on multiple CPU cores concurrently. Furthermore, each I/O Handler can be bounded to a dedicated CPU core to reduce the performance overhead caused by context switches.

3.3 I/O Request Processing

In NV-BSP, each I/O request is processed in two phases. In the first phase, I/O request is distributed into I/O handler command queue according to its target logical address region. In the second phase, the corresponding I/O handler processes the request based on the RAID request handling tree model, as shown in Fig. 4. The I/O request processing acts in accordance with the tree changes. The tree grows when an I/O request arrives at the corresponding VDisk and the tree shrinks as I/O requests being served successfully.

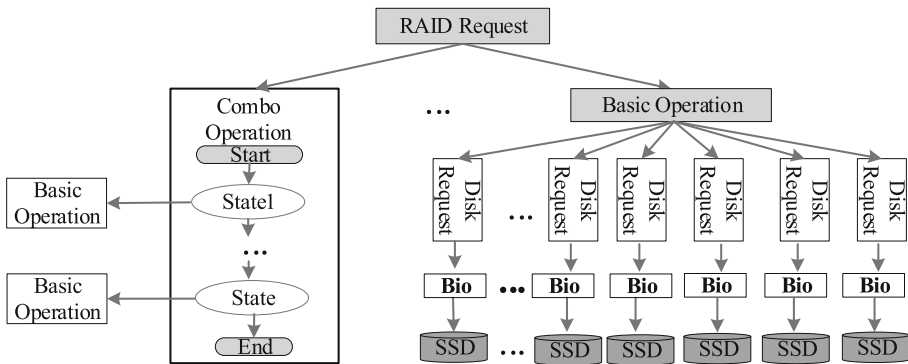


Fig. 4. I/O request processing in NV-BSP.

All the I/O operations to NV-BSP can be divided into two categories, i.e., basic operation and combo operation. A basic operation only involves in a single stripe and can be easily accomplished. A combo operation consists of one or more basic operations. For example, for the reconstruction combo operation, it consists of two basic operations, i.e., read a stripe to memory and write the reconstructed data from memory to the spare space. Each combo operation corresponds to a sequentially executed state machine. The combo operation is converted into multiple basic operations and executed recursively according to the state machine until all the basic operations are accomplished.

4 Performance Evaluation

In this section, we evaluate the detailed behavior of NV-BSP under synthetic FIO workloads.

4.1 Experimental Setup

The experimental setup consists of a server with two Intel Xeon Gold 6128 CPUs (each with 6 physical cores and 12 logical cores in Hyper-Threading mode), 192 GB of DDR4 RAM, 8 NVMe SSDs with 1.8 TB NAND Flash. The operating system is CentOS Linux 7.7 with the kernel version of 4.19.46. NV-BSP is implemented as a kernel module in Linux and rely on FIO of version 3.7 for performance evaluation. In the following experiments, all the workloads generated by FIO use Linux Asynchronous I/O (libaio) engine and enable direct IO.

4.2 Experiment Results

Performance Scalability Measurement. To understand the system performance of NV-BSP with a different number of NVMe SSDs, we measure the IOPS of the VDisks from storage arrays equipped with 3 to 8 NVMe SSDs. In this experiment, we configure FIO workloads to 4 KB read-intensive (30% write and 70% read) and write-intensive (70% write and 30% read) I/O, 8 threads with 64 queue depth per thread.

Figure 5 shows the comparison of the IOPS of VDisks from the NV-BSPs with a different number of NVMe SSDs. NV-BSP generally achieves obvious performance improvement for both read-intensive and write-intensive workloads as more NVMe SSDs are equipped in NV-BSP. Specifically, for the read-intensive workload, IOPS improves 86.94% when the number of NVMe SSDs increases from 3 to 6, and 97.82% when the number of NVMe SSDs increases from 4 to 8. For the write-intensive workload, IOPS improves 87.04% when the number of NVMe SSDs increases from 3 to 6, and 97.98% when the number of NVMe SSDs increases from 4 to 8.

To further evaluate the performance benefits from the high concurrent I/O processing design in NV-BSP, we configure a different number of I/O handlers in a VDisk from an NV-BSP equipped with 8 NVMe SSDs and measure both the IOPS under 4 KB random read/write and the bandwidth under 128 KB sequential read/write workloads of the VDiks.

As shown in Fig. 6, both random read and random write performance of the VDisk improves linearly until the number of I/O handlers increases to 16, where the 4 KB random performance of the VDisk reaches the bottleneck and appears little improvement. Figure 7 depicts the sequential read and write bandwidth of the VDisks with a different number of I/O handlers. Similar to the random read and write performance, NV-BSP shows a good acceleration ratio with the increasing number of I/O handlers, especially for the sequential read performance. We can clearly see that the maximum read bandwidth of a single VDisk can be more than 20 GB/s, which indicates outstanding performance scalability of NV-BSP.

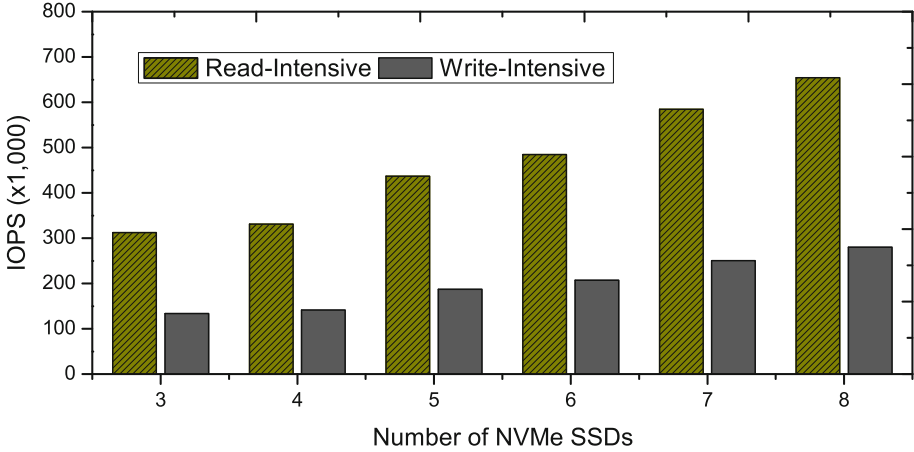


Fig. 5. IOPS comparison for different number of NVMe SSDs in NV-BSP.

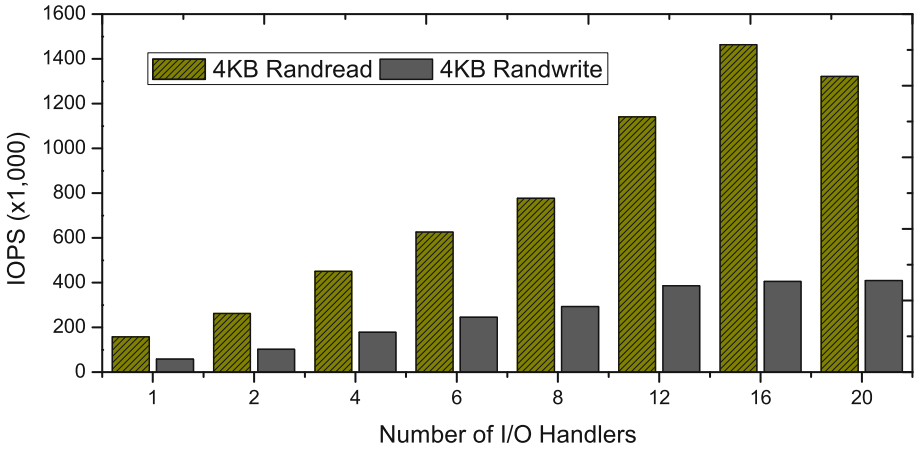


Fig. 6. IOPS comparison for different number of I/O Handlers in NV-BSP.

Performance Comparison with MD-RAID. We compare the performance of NV-BSP with the build-in MD-RAID in Linux. Two different VDIs created by NV-BSP and MD-RAID are evaluated in this experiment. Figure 8 and Fig. 9 show the average I/O latency and bandwidth comparisons for different I/O request sizes respectively. As shown in Fig. 8, the average I/O latency of NV-BSP is much lower than that of MD-RAID. As the I/O request size increases from 4 KB to 128 KB, the average I/O latency of NV-BSP is at least 4.75x lower than that of MD-RAID. Similarly, the bandwidth of the VDisk from NV-BSP is much higher than that of MD-RAID. As the I/O request size increases from 4 KB to 128 KB, the bandwidth of NV-BSP is 1.64x to 11.06x higher than that

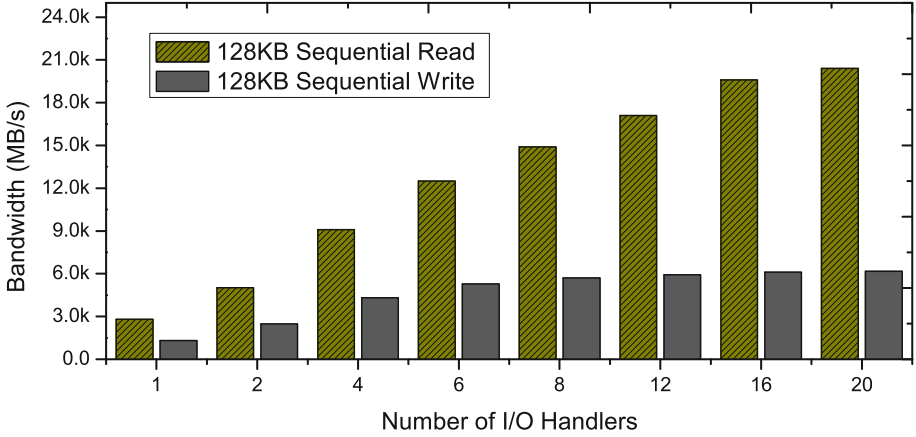


Fig. 7. Bandwidth comparison for different number of I/O Handlers in NV-BSP.

of MD-RAID. Apparently, both the I/O latency and bandwidth confirm that the performance of NV-BSP significantly outperforms that of MD-RAID.

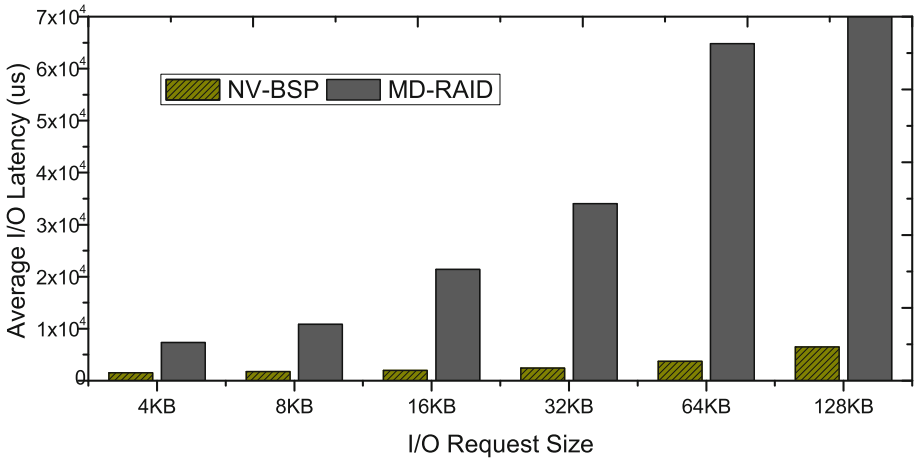


Fig. 8. Latency comparison for different I/O request sizes.

5 Related Work

The increasing number of CPU cores in modern storage servers enables a single storage server to host a large amount of high-performance NVMe SSDs [5, 8]. However, the scalability issues of storage software stack significantly prevent

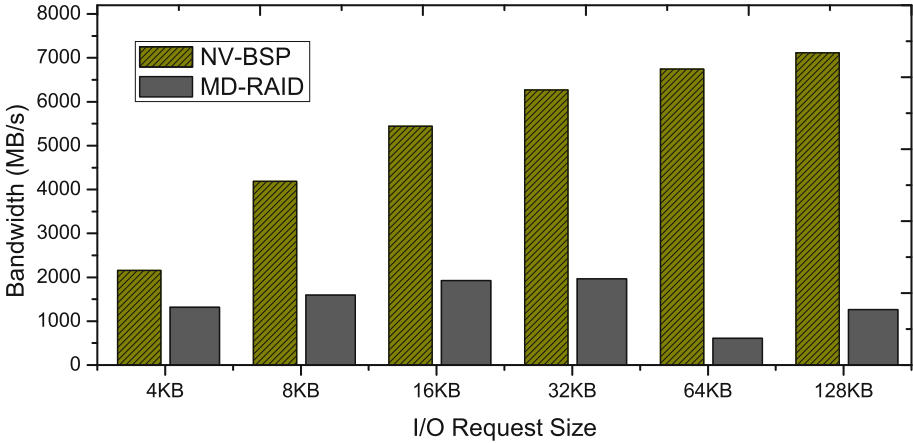


Fig. 9. Bandwidth comparison for different I/O request sizes.

the output performance of the NVMe SSDs on a server. Several studies have tried to sequentially write data into the storage array to improve system performance and achieve wear-leveling among NVMe SSDs [11]. SWAN [7] introduces a log-structured storage management logic at the host level to sequentialize the written data to RAID stripes. SWAN is specially designed for NVMeoF target by balancing the output performance of NVMe SSDs and network interface performance. Purity [3] developed by Pure Storage proposes to adopt an LSM-tree [12] based log-structured indexes and data layouts in storage array, thus data can be written in large sequential chunks for better performance. Besides, purity also integrates compression and deduplication to make better use of NVMe SSD capacity. Different from purity and SWAN, NV-BSP proposes to alleviate the overhead from lock contentions and context switches in the critical I/O path to achieve the high concurrent I/O processing.

Integrating flash storage disaggregation techniques (e.g., iSCSI and NVMeoF) into storage array designs shows great benefits of improving storage resource utilization and aggregating the performance of a bunch of storage devices [1, 9]. Storage array manufacturers like Huawei, Pure Storage, Apeiron Data Systems, Kamnario, Pavilion Data Systems have released their storage array products and solutions that integrate NVMeoF target logic respectively. For example, OceanStor Dorado V6 uses NVMe over FC and RDMA as the front end interfaces of RAID 2.0 storage pools. E8 Storage combines the high performance of NVMe drives, the high availability and reliability of centralized storage, and the high scalability of scale-out solutions in a single storage array. The E8-D24 and E8-S10 products of E8 Storage can deliver up to 10 million IOPS with 40 GBps throughput using 100 GbE or 100 Gbps InfiniBand connectivity.

6 Conclusion

This paper presents NV-BSP, an NVMe SSD-based Burst I/O Storage Pool that leverages the performance benefits of NVMe SSD, NVMeoF Protocol, and RDMA networks in HPC storage systems. NV-BSP disaggregates NVMe SSDs from HPC compute nodes to enhance the scalability of HPC storage systems, employs fine-grained chunks rather than physical NVMe SSD devices as the RAID-based data protection areas, and exploits high concurrent I/O processing model to alleviate the performance overhead from lock contentions and context switches in critical I/O path. We evaluated and analyzed the detailed behavior of NV-BSP. Compared with node-local SSDs in HPC, NV-BSP provides a full system solution of storage disaggregation, delivers comparable performance, and significantly improves system reliability. Compared with the built-in MD-RAID in Linux, NV-BSP achieves much better system performance. In future work, we will study global wear-leveling in NV-BSP to enhance the endurance of NVMe SSDs and QoS management scheme to create VDisk from NV-BSP with customized capacity and performance.

Acknowledgment. The authors would like to thank the anonymous reviewers. This work was supported in part by the Advanced Research Project of China under grant 31511010202 and the National Key Research and Development Program of China under Grant 2018YFB0204301.

References

1. Amvrosiadis, G., et al.: Data storage research vision 2025: report on NSF visioning workshop held May 30–June 1, 2018. Technical report, USA (2018)
2. Balakrishnan, M., Kadav, A., Prabhakaran, V., Malkhi, D.: Differential raid: rethinking raid for SSD reliability. *ACM Trans. Storage* **6**(2), 1–22 (2010). <https://doi.org/10.1145/1807060.1807061>
3. Colgrove, J., Davis, J.D., Hayes, J., Miller, E.L., Sandvig, C., Sears, R., et al.: Purity: building fast, highly-available enterprise flash storage from commodity components. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015*, pp. 1683–1694. Association for Computing Machinery, New York (2015). <https://doi.org/10.1145/2723372.2742798>
4. Guz, Z., Li, H.H., Shayesteh, A., Balakrishnan, V.: NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In: *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR 2017*. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3078468.3078483>
5. Jackson, A., Turner, A., Weiland, M., Johnson, N., Perks, O., Parsons, M.: Evaluating the arm ecosystem for high performance computing. In: *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2019*. Association for Computing Machinery, New York (2019). <https://doi.org/10.1145/3324989.3325722>
6. Jin, Y.T., Ahn, S., Lee, S.: Performance analysis of NVMe SSD-based all-flash array systems, pp. 12–21 (2018)

7. Kim, J., Lim, K., Jung, Y., Lee, S., Min, C., Noh, S.H.: Alleviating garbage collection interference through spatial separation in all flash arrays, pp. 799–812 (2019)
8. Kim, J., Ahn, S., La, K., Chang, W.: Improving I/O performance of NVMe SSD on virtual machines. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC 2016, pp. 1852–1857. Association for Computing Machinery, New York (2016). <https://doi.org/10.1145/2851613.2851739>
9. Klimovic, A., Litz, H., Kozyrakis, C.: Reflex: remote flash local flash. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, pp. 345–359. Association for Computing Machinery, New York (2017). <https://doi.org/10.1145/3037697.3037732>
10. Liao, X., Xiao, L., Yang, C., Lu, Y.: Milkyway-2 supercomputer: system and application. *Front. Comput. Sci.* **8**(3), 345–356 (2014)
11. Oh, Y., Choi, J., Lee, D., Noh, S.H.: Improving performance and lifetime of the SSD raid-based host cache through a log-structured approach. In: Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW 2013. Association for Computing Machinery, New York (2013). <https://doi.org/10.1145/2527792.2527795>
12. Oneil, P., Cheng, E.Y.C., Gawlick, D., Oneil, E.: The log-structured merge-tree (LSM-tree). *Acta Informatica* **33**(4), 351–385 (1996)
13. Patel, T., Byna, S., Lockwood, G.K., Tiwari, D.: Revisiting I/O behavior in large-scale storage systems: the expected and the unexpected (2019)
14. Qian, J., Jiang, H., Srisa-An, W., Seth, S., Skelton, S., Moore, J.: Energy-efficient I/O thread schedulers for NVMe SSDs on NUMA. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2017, pp. 569–578. IEEE Press (2017). <https://doi.org/10.1109/CCGRID.2017.24>
15. Shi, X., Liu, W., He, L., Jin, H., Li, M., Chen, Y.: Optimizing the SSD burst buffer by traffic detection. *ACM Trans. Archit. Code Optim.* **17**(1), 1–26 (2020). <https://doi.org/10.1145/3377705>
16. Tavakkol, A., et al.: Flin: enabling fairness and enhancing performance in modern NVMe solid state drives. In: Proceedings of the 45th Annual International Symposium on Computer Architecture, ISCA 2018, pp. 397–410. IEEE Press (2018). <https://doi.org/10.1109/ISCA.2018.00041>
17. Xie, X., Li, Q., Wei, D., Song, Z., Xiao, L.: ECAM: an efficient cache management strategy for address mappings in flash translation layer. In: Wu, C., Cohen, A. (eds.) APPT 2013. LNCS, vol. 8299, pp. 146–159. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45293-2_11
18. Xie, X., Wei, D., Li, Q., Song, Z., Xiao, L.: CER-IOS: internal resource utilization optimized I/O scheduling for solid state drives. In: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pp. 336–343. IEEE (2015)
19. Xie, X., Xiao, L., Wei, D., Li, Q., Song, Z., Ge, X.: Pinpointing and scheduling access conflicts to improve internal resource utilization in solid-state drives. *Front. Comput. Sci. Chin.* **13**(1), 35–50 (2019)
20. Xie, X., Yang, T., Li, Q., Wei, D., Xiao, L.: Duchy: achieving both SSD durability and controllable SMR cleaning overhead in hybrid storage systems. In: Proceedings of the 47th International Conference on Parallel Processing, p. 81. ACM (2018)

21. Xu, G., et al.: RFPL: a recovery friendly parity logging scheme for reducing small write penalty of SSD raid. In: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019. Association for Computing Machinery, New York (2019). <https://doi.org/10.1145/3337821.3337887>
22. Xu, Q., et al.: Performance characterization of hyperscale applications on on NVMe SSDs. In: Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2015, pp. 473–474. Association for Computing Machinery, New York (2015). <https://doi.org/10.1145/2745844.2745901>
23. Xu, Q., et al.: Performance analysis of NVMe SSDs and their implication on real world databases. In: Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015. Association for Computing Machinery, New York (2015). <https://doi.org/10.1145/2757667.2757684>
24. Zhang, B., Yang, M., Xie, X., Du, D.H.C.: Idler: I/O workload controlling for better responsiveness on host-aware shingled magnetic recording drives. *IEEE Trans. Comput.* **69**(6), 777–788 (2020)



Pin-Tool Based Execution Backtracking

Shuangjian Wei¹, Weixing Ji¹(✉), Qiurui Chen², and Yizhuo Wang¹

¹ Beijing Institute of Technology,
Beijing 100081, China
{sjw,jwx,frankwy}@bit.edu.cn

² Science and Technology on Special System Simulation Laboratory,
Beijing Simulation Center, Beijing 100854, China
qiuruich@126.com

Abstract. Checkpoint/restart is a common fault tolerant technique which periodically dump state to reliable storage and restart applications after failure. Most of existing checkpoint/restart implementations only handle volatile state and lack of support for persistence state of applications. Even the algorithm specifically designed for file checkpointing may not support complex operations and some need to modify source code. This paper presents a new checkpoint technique, which use dynamic instrumentation to temporarily cache disk operations in memory, and use existing memory checkpoint tool to dump or restore process state at runtime. We show that not only can this method create regular checkpoints for both volatile and persistence state, but also has important applications in execution backtracking.

Keywords: Checkpointing · Dynamic instrumentation · Execution backtracking · Volatile state · Persistence state

1 Introduction

Checkpoint/Restart (C/R) is a mainstream fault-tolerant technique. It generates checkpoints periodically to save execution state and recovers from checkpoints after process fails. The behavior of a process has three parts: volatile data, persistent data, and OS environment [1]. Among them, volatile data refers to data in memory and registers, and they are lost after power-off. Persistent data refers to the data stored in stable storage, such as files and databases. OS environment refers to the resources that user processes must access at runtime, such as swap space and monitors. In this paper, we focus on the C/R of both volatile data and persistent data.

The consistency of volatile and persistent data is a prerequisite for process restart. Unlike incorrect recovery of volatile state (which usually leads to obvious process failures), incorrect rollback of persistent state usually leads to more serious losses due to difficulty in tracing, so it has become a major concern for many users. Unfortunately, most existing mainstream checkpoint tools do not support or do not fully support file checkpoints. Fault-tolerant systems that use

such tools roll back the volatile state to the previous checkpoint after a process fails, while keeping the file state unchanged. If the process has modified these files, such as writing, deleting, renaming, etc., it will cause erroneous results. There are many types of these errors, and the following figures show two common cases.

```

checkpoint i;
fd = open("doc",
           O_WRONLY|O_APPEND);
write(fd, buf, buf_size);
/* failure occurs */
checkpoint i+1;

```

Fig. 1. A process writes same data multiple times.

```

fd = open("doc", O_RDWR);
checkpoint i;
read(fd, buf, buf_size);
lseek(fd, 0, SEEK_SET);
write(fd, buf, buf_size);
/* failure occurs */
checkpoint i+1;

```

Fig. 2. A process reads dirty data and causes error.

In Fig. 1, the program opens the file “doc” after checkpoint i , and writes data at the end of the file in appending mode, then an error occurs right before checkpoint $i+1$. During the rollback, since the information of “doc” is not recorded in checkpoint i , the rollback algorithm will not truncate “doc”, so that the content will be added again after the execution is resumed. In Fig. 2, the program performs the read-before-write operation at the same position on the “doc” after checkpoint i , and then an error occurs before checkpoint $i+1$. During the rollback, because there is no “doc” information in checkpoint i , the rollback algorithm will not restore the file state, which causes the program to read dirty data after recovery. The above two errors are also known as RARE (Rollback After Real time Event) and RARW (Rollback After Reading and Writing the same area) [2].

Over the past 20 years, many checkpoint algorithms and tools have been proposed. These algorithms and tools play an irreplaceable role in tasks such as scheduling management, process migration, and load balance. Nonetheless, checkpoint related work is far from over, especially in the field of file checkpoints. There are three main shortcomings in existing file checkpoint algorithms:

- Only idempotent operations and very few non-idempotent operations are supported. Idempotent operations include all operations that do not change the consistency state of the file, such as read. User applications that use this type of checkpoint tool can only access files in read-only and appending modes.
- Only active files are supported. Active files are those files that were open at the time the checkpoint was created. Such tools traverse all open file handles and record the current file length when a checkpoint is created.
- The user application source code must be modified to fit the file checkpoint feature. Currently, most of the file checkpoint tools are provided as libraries, and they are implemented by encapsulating file interfaces. User programs that

have been built must modify the source code to accommodate these libraries. This method will not only increase the workload of developers, but also leave security risks for the system.

Although kernel-level checkpointing tools can address all of these issues, they can also introduce significant overhead for applications that do not require checkpointing. This article mainly has the following three contributions. First, this article introduces a new C/R technique that neither modifies program source code nor restricts process file access operations. Secondly, a method to dump the complete state of the process using only the memory checkpoint tool is proposed. Finally, the checkpoint method proposed in this paper not only can set up regular checkpoints, but also assist in execution-backtracking, that is, roll back the process state to any point in the execution history.

In addition to this section, application scenarios are discussed in Sect. 2 and related work is given in Sect. 3. The architecture overview is introduced in Sect. 4, and Sect. 5 presents system implementation details. The evaluation is in Sect. 6 and Sect. 7 is the summary.

2 Application Scenario

To describe the usage scenarios of the ideas presented in this article, it is necessary to first explain how the existing checkpoint tools work. Existing checkpoint tools, such as MOB [2, 16] and CprFS [20] mentioned in the following section, will automatically create process checkpoints at regular intervals. When the process execution fails, the system will automatically select the most recent checkpoint, such as checkpoint i in Fig. 1 and Fig. 2, and restart the program from this checkpoint. Restarting the process from checkpoint i discards all changes made to the file during erroneous execution and minimizes work loss, which is also the ultimate purpose of checkpoint tools. Although multiple checkpoints are set during process execution, only one checkpoint (checkpoint i) is involved in the entire C/R process. The process cannot roll back the state to checkpoint $i - 1$ or $i - 2$, because all file modification data before checkpoint i has been discarded when setting checkpoint i . Therefore, we can conclude that the existing checkpoint tool is to ensure that the target process can be safely and error-freely executed to the end in one execution. Only one checkpoint (the most recent checkpoint) is required to ensure the execution of the process.

Unlike the existing checkpoint tools for program fault tolerance, the checkpoint method proposed in this paper can also be used for execution backtracking. Execution backtracking refers to the operation of rolling back the process state to any moment in its execution history. Taking the simulation programs as an example, in order to obtain the simulation results under different parameters, users need to execute the same simulation program multiple times and enter different parameters for it. To reduce the time it takes to re-execute, we can set a checkpoint before setting the parameters and restart the process from the checkpoint in the next execution. It should be emphasized that the program started from the checkpoint can also set the checkpoint again. These checkpoints will

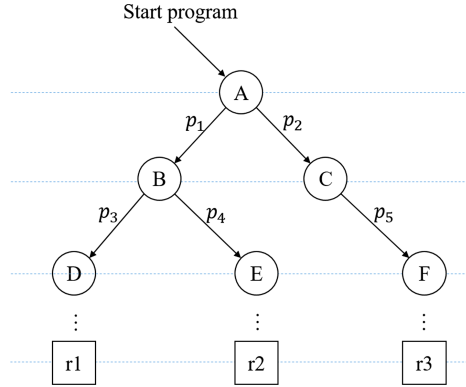


Fig. 3. Tree structure formed by checkpoints

eventually form a tree structure (see Fig. 3), and the process can be restarted and executed from any node in the tree. The non-leaf nodes in the tree in Fig. 3 refer to checkpoints, and the leaf nodes represent program execution results. The strategy proposed in this paper can not only ensure the correct execution of the process, but also meet the requirements of process traceback, that is, restart the program from any checkpoint in the checkpoint tree.

3 Related Work

3.1 Checkpointing

The BLCR (Berkeley Lab’s Linux Checkpoint/Restart project) presented in [3–7] is a robust kernel-level checkpoint/restart implementation that can support a variety of parallel scientific codes. In terms of file processing, BLCR only records the file size when creating a checkpoint and simply truncates the file to its original length during recovery. Although this strategy is very lightweight and effective in log-only scientific calculations, it is not applicable in practical applications.

The ftIO system [8] is implemented by encapsulating the standard file interface. In order to avoid file access errors between two adjacent checkpoints, ftIO has designed a new file access protocol. This protocol is based on the copy-on-write [9] concept, where the entire file is copied upon the first write operation. Subsequent file operations are performed on the replica. During checkpointing, the modifications are committed by simply replacing the original file with its replica. The ftIO algorithm is concise and effective, but it introduces a huge time and space overhead when processing large files, which can seriously drag down user processes.

The core idea of Libckpt mentioned in [1, 10, 11] is to use lazy-coordination and shadow copy to solve the problem of inactive files. The basic concept of lazy coordination is that file data is not processed immediately when a checkpoint is

created, but is deferred until the file content actually changes. Just record the file size when the file becomes active, and make a shadow copy of the file when the file content is about to be modified. Libckpt’s strategy to reduce runtime and space overhead is to perform shadowing page by page.

Libfcp [12] uses in-place updates [13] with undo logs to checkpoint files. It intercepts all file operations except read-only files through the encapsulated file interface. When a file is opened for modification, the size of the file is recorded and a truncated undo log of the file is generated. When the contents of a file are modified, it generates an undo log that restores the contents of the file. Libfcp_RM [14] enhances Libfcp by adding transaction management. It uses transactions to atomize a sequence of file updates in the application. Libra [15] combines a “copy-on-change” strategy with an undo log to keep track of what really changed to reduce the log size.

The MOB (Modification Operation Buffer) mentioned in [2, 16] buffers all modification operations after one checkpoint until the next checkpoint, so that all operations between two checkpoints become atomic. MOB’s basic buffering strategy is to append new content directly after the existing buffer. If the same area of the file is modified more than twice, the buffer will not append new content, but update the original data in the buffer. MOB transfers the execution of file operations to memory, which can significantly reduce file access time overhead. In addition, MOB uses a disk buffer to limit the amount of memory occupied by the algorithm.

The VFO (Virtual File Operation) proposed in [17] buffers all the write operations after a checkpoint until the next one, making all the operations between two checkpoints atomic. The read and write operations of the user process do not directly interact with the disk file, but access to the virtual file operation management table entries, just like inserting a virtual file layer between the user and the file. Unlike MOB, VFO manages file data in blocks, reducing space overhead. Metamori [18] is another MOB-like file checkpointing algorithm. It adds support for file streams on top of MOB, which only supports file descriptors. In addition, it also optimizes the related data structure and uses a B-tree to manage the buffer mapping table to improve retrieval efficiency.

CprFS [20] uses the FUSE [21] module in the Linux system to create a file system that executes in user space. For checkpoint, an atomic transaction is considered to be the execution of a program between two consecutive checkpoints. The program either commits its state during checkpointing or aborts at some point during execution, in which case it can be recovered from the last checkpoint. CprFS has high execution efficiency, and does not need to modify the program source code.

3.2 Execution Backtracking

Execution backtracking is the process of restoring the state of a program to any earlier point in its execution history. It is used to facilitate program debugging. The Spyder mentioned by [26] is a system for selective checking of computational sequences. It allows users to step back from the checkpoint without having to

re-execute the program to reach the most recent previous state. [27] describes a debugging method that uses a combination of re-execution and backtracking to find the first difference in the calculation, which may eventually lead to incorrect values indicated by the user. [28] provides a debugging model based on dynamic program slicing and execution backtracking technology that easily lends itself to automation.

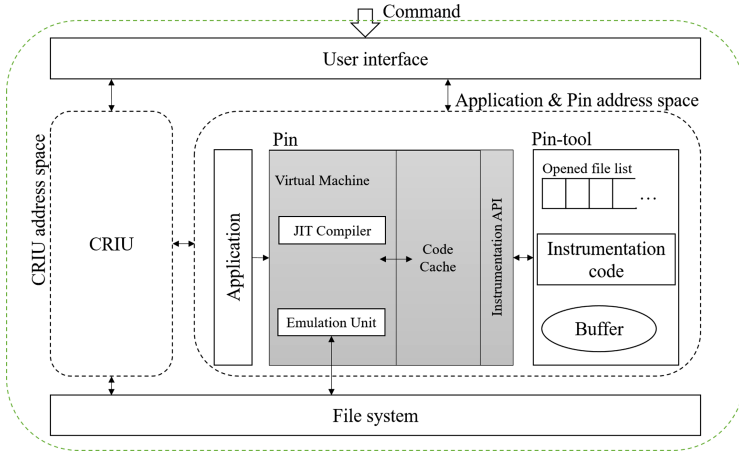


Fig. 4. System architecture diagram

4 Architecture Overview

The traceback system architecture is shown in Fig. 4, where the user interface refers to the client of the system, which displays system status information to users and receives instructions from the users. In addition to this, the user interface is also responsible for managing the tree formed by checkpoints, as well as issuing commands to the checkpoint tool and inputting parameters to the user process. Users can issue checkpoint setting commands through the interface, or select a checkpoint from the tree to restart a process. The checkpoint tool used in the system is CRIU (Checkpoint/Restore In Userspace) [22], which is an open source software on GitHub. CRIU works on the Linux operating system. It can freeze the target process after receiving user commands, and then dump the process data to disk. CRIU completes the checkpoint restarting process by transforming itself into a task to be restored.

Pin [25] allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The best way to think about Pin is as a “just in time” (JIT) compiler. The input to this compiler is not bytecode, however, but a regular executable. Here, we use Pin to build a tool (Pin-tool in Fig. 4) for intercepting and caching file operations. Pin-tool is mainly composed of three parts,

a list for managing open files, a series of instrumentation codes for intercepting file operations, and a buffer for buffering file contents. The three components of Pin-tool form a virtual file layer, which can load the contents of the disk file into the buffer, and can also transparently cache the data written by the user process. After CRIU receives the user's request to set a checkpoint, it directly dumps the volatile data of Pin-tool and user application to disk. Because the file modification information is stored in the virtual file layer, each checkpoint created by CRIU contains all the information of the process.

5 Implementation

The disk file always remains the same to ensure that the process can be correctly executed back to any point in the execution history. We use the virtual file layer mentioned in the previous section to buffer all file changes made by the process. Unlike existing methods, we use dynamic instrumentation to intercept and replace the corresponding functions in the process, thereby avoiding the work and risks caused by modifying the source code. Instrumentation is to insert some probes into the program to collect the tested program information on the basis of ensuring the original logical integrity of the tested program. These probes are essentially code segments for information collection, which can be function calls that distribute information or collect information. The code is added dynamically while the executable is running. The functions we intercept and replace include: **open**, **close**, **read**, **write**, **create**, **dup**, **dup2**, **dup3**, **fcntl**, **lseek**, **remove**, etc.

Table 1. Data structure for storing file information.

Name	Description	Name	Description
<code>_fd</code>	File descriptor	<code>_flags</code>	File access mode
<code>_path</code>	File path	<code>_closed</code>	Whether the file is closed
<code>_flags_real</code>	Real file access mode	<code>_pos_wd</code>	File write pointer
<code>_pos_rd</code>	File read pointer	<code>_len_acc</code>	File accessible length
<code>_len_cur</code>	Current file length	<code>_pages</code>	File content buffer

5.1 Data Structure

For each opened file, a global data structure is created to store its information. We call this data structure **FileEntry**. As shown in Table 1, `_fd` and `_flags` represent the file descriptor and file open mode, respectively. `_path` refers to the file path, `_closed` is used to describe whether the file has been closed. To ensure that the disk file remains unchanged, the virtual file layer will change the file open

mode and record the mode in `_flags_real`. The following `_pos_rd` and `_pos_wd` are file read and write pointers, while `_len_acc` and `_len_cur` are the file's accessible length and current length. The open mode of the file will affect the value of `_len_acc`, and the process of writing will change the value of `_len_cur`. The virtual file layer manages the file contents in pages and loads the corresponding pages into `_pages` when needed.

5.2 Virtual File Layer

This section discusses how to transparently perform file operations in the buffer. The virtual file layer is composed of a set of file access functions and a buffer. The main purpose of its existence is to unify memory and disk file so that memory checkpoint tool dumps all process data. The way it works is to intercept all file operations performed by the process and transparently execute the operations in the buffer, so the dynamically inserted code has a completely different role from the native code.

Algorithm 1. New file open function

```

1: function NEWOPEN(filename, flags, mode)
2:   if flags = O_RDONLY then return open(filename, flags, mode)
3:   end if
4:   if FIND(filename, fety) then return CHANGE_MODE(fety)
5:   end if
6:   if access(filename, F_OK) = -1 And (flags & O_CREAT) then
7:     open(filename, O_CREAT, mode)
8:   end if
9:   fety._fd ← open(filename, O_RDONLY)
10:  fety._closed ← false
11:  fety._path ← filename
12:  fety._flags ← flags
13:  fety._pos_rd, fety._pos_wd ← 0
14:  if flags & O_TRUNC then
15:    fety._len_acc ← 0
16:    fety._len_cur ← 0
17:  else
18:    fety._len_acc ← lseek(fety._fd, 0L, SEEK_END)
19:    fety._len_cur ← fety._len_acc
20:  end if
21:  if flags & O_APPEND then fety._pos_wd ← fety._len_cur
22:  end if
23:  g_files_vec.push_back(fety)
24:  return fety._fd
25: end function

```

Each file opened by the user process has an independent buffer containing multiple fixed-size pages. The file just opened by the process does not load any data

into the buffer, and data loading is delayed until the file is actually written or read. To avoid memory overflow caused by excessive file size, the file data always loaded in pages. The file read/write pointer and read/write size jointly determine which page needs to be loaded immediately. The virtual file layer does not handle read-only files, because even if the instrumentation code does nothing, the disk file will not change.

NewOpen is a function for replacing **open** in the virtual file layer. It is used to transparently open a file and return the file descriptor after recording the file information. The virtual file layer does not record any information about files opened in read-only mode, and directly calls **open** and returns the result. For a newly opened file, create a FileEntry instance (*fety*) and initialize its contents according to the open mode and file status, and finally insert it into the global list. Its detailed description is shown in Algorithm 1, where *g_files_vec* is the global list used to store information about all open files. The **Find** function is responsible for finding the current file information in *g_files_vec* to determine whether the file was previously opened. For the file that has been opened, the virtual file layer no longer creates a new FileEntry instance, but uses the **ChangeMode** function to change the file information based on the existing *fety*.

NewClose is a function for replacing **close** in the virtual file layer, and is responsible for closing the opened file descriptor. For files present in *g_files_vec*, first set the **_closed** flag to true, then close the file descriptor. The reason why the data of the closed file is not deleted is that the user process may reopen the file in the subsequent execution.

Algorithm 2. New file read function

```

1: function NEWREAD(fd, buf, count)
2:   if FIND(fd, fety) then return fety.READFROMPAGES(buf, count)
3:   else
4:     return read(fd, buf, count)
5:   end if
6: end function
7: function READFROMPAGES(buf, count)
8:   page_str ← _pos_rd/PAGE_SIZE
9:   page_end ← (_pos_rd + count - 1)/PAGE_SIZE
10:  read_num ← 0
11:  for i = page_str → page_end do
12:    LOADONEPAGE(i)
13:    read_num += READFROMONEPAGE(buf + read_num, count - read_num)
14:  end for
15:  return read_num
16: end function

```

NewRead is a function for replacing **read** in the virtual file layer to read a certain number of bytes and return the number of bytes read. When **NewRead**

is called, it first obtains the handle **fety** used to manipulate the file. If **fety** does not exist, it directly calls native **read** and return. As shown in lines 9 and 10 of Algorithm 2, using the `_pos_rd` of the current file and the parameter `count` can calculate the pages that may need to be loaded. Then use **LoadOnePage** and **ReadFromOnePage** to load and read out the data in the file (as shown in lines 12 to 17 of Algorithm 2), and finally return the number of bytes read. The judgment of the file boundary (`_len_cur`) and the change of the read pointer (`_pos_rd`) are made in **ReadFromOnePage**. The return value may be less than `count` when touching the file boundary. Because the preset page size is often much larger than the read size, the read operation after a page loads will be much faster than reading directly from the file. The page loaded in the read operation will also speed up the program write operation.

Algorithm 3. New file write function

```

1: function NEWWRITE(fd, buf, count)
2:   if FIND(fd, fety) then return fety.WRITETOPAGES(buf, count)
3:   else
4:     return write(fd, buf, count)
5:   end if
6: end function
7: function WRITETOPAGES(buf, count)
8:   page_str  $\leftarrow$  _pos_wd/PAGE_SIZE
9:   page_end  $\leftarrow$  (_pos_wd + count - 1)/PAGE_SIZE
10:  write_num  $\leftarrow$  0
11:  for i = page_str  $\rightarrow$  page_end do
12:    LOADONEPAGE(i)
13:    write_num += WRITETOONEPAGE(buf + write_num, count - write_num)
14:  end for
15:  return write_num
16: end function

```

The execution flow of the **NewWrite** function used to replace **write** is similar to **NewRead**, and the corresponding page needs to be loaded before writing. The difference is that if the page to be loaded does not exist, **LoadOnePage** will create a blank page instead of doing nothing for writing new data. Unless the memory overflows or other errors occur, the return value is always the same as `count`.

The core idea of this paper is to use memory buffer file operations to unify volatile and persistent data so that the memory checkpoint tool can dump all the data of the process. The advantage of this strategy is that it is simple and effective for processes that have sufficient memory space or access to files that are not too large, and will not negatively affect the execution speed. But for large files, it may cause a shortage of memory space. It is unrealistic to completely buffer the contents of larger files into memory. If the file accessed by a process is too large, the virtual file layer will write back buffered data to the disk, and at the same time create a file backup on the disk for process backtracking.

In addition to the above four basic file operations, the virtual file layer also supports operations such as **remove**, **rename**, and **redirect**. When the user program calls the **remove** function, the virtual file layer will first close the corresponding file descriptor, then release the file buffer, and finally set the file accessible length to 0. For the rename operation of the user process, the virtual file layer will change the value of `_path` in the file entry. User process redirection operations, such as **freopen**, **dup**, **dup2**, **dup3**, etc., will cause the original file descriptor to be closed and create a new file descriptor (or use the specified file descriptor) instead.

6 Evaluation

In this section, we use micro-benchmarks and real-world applications to evaluate our method to prove that dynamic instrumentation and checkpoint overhead are tolerable. The experiment was conducted on a computer with Intel(R) Core(TM) i7-8550U CPU @ 1.80 GHz, 4 GB RAM and a 20 GB disk space. The operating system used was CentOS-7 with kernel 3.10.0-693.el7. The file system for local disk was xfs.

6.1 IOzone Test

How to dump files is the core problem to be solved in process backtracking. The method used in this paper is to insert a virtual file layer between the disk and process through dynamic instrumentation technology. The access speed of the process to the file, especially the speed of writing the file is closely related to the execution speed of the program. We use IOzone [23] to evaluate the execution efficiency of instrumentation code. The experiment uses the original IOzone and the IOzone after dynamic instrumentation to write 1GB data in different block sizes, and then records the writing speed. The experimental results are shown in the figure (see Fig. 5).

The experiment tested the write speed of the xfs file system in different states, where xfs-a and xfs-b respectively represent the write speed of the file system with and without calculating the flush time. The black bar shows the writing speed of the file system after dynamic instrumentation. From the data shown in the figure, we can find that the file access speed after dynamic instrumentation is similar to the native file system, and in most cases is slightly higher than the native file system. Therefore, we believe that the impact of the new virtual file layer on the simulation program is positive, as can be seen from the total time of the simulation program execution in the previous section.

6.2 Pin-Tool Overhead

An important part of the implementation of the backtracking strategy based on checkpoints is dynamic instrumentation. Dynamic instrumentation allows developers to intercept or replace existing methods in the original program without

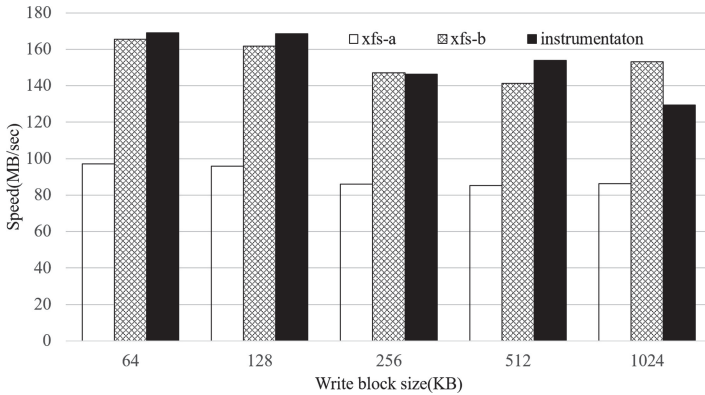


Fig. 5. Write speed of virtual file layer

changing the source code. Obviously, this requires additional memory overhead. In order to detect memory overhead, we instrument the existing program and then sample during the program execution. The test program we use is BWA [24], which is a software package for mapping DNA sequences against a large reference genome (such as the human genome). We can find its source code on GitHub. Figure 6 shows the memory overhead information of the process using 9 samples.

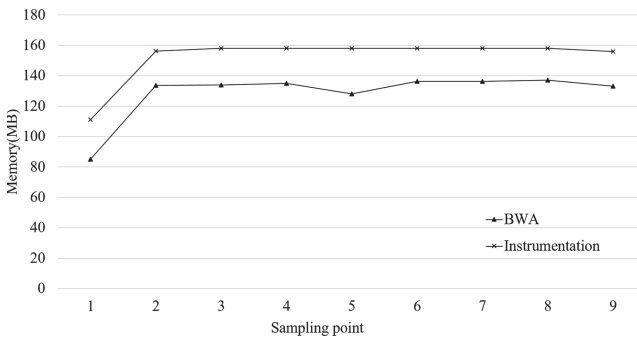


Fig. 6. Memory overhead caused by dynamic instrumentation.

The upper and lower two lines in the figure present the memory occupation trend of the BWA program after dynamic detection and the original BWA program during execution. Since the memory usage of the BWA program in the steady state does not change much during execution, the lower line is almost horizontal. Correspondingly, the memory overhead of the BWA program in the stable state after dynamic instrumentation is also displayed as a horizontal state, which indicates that the memory overhead caused by dynamic instrumentation is an approximately fixed value and does not change with the process size and

execution status. The gap between the upper and lower lines (the difference is about 30M) is the overhead caused by dynamic insertion. This fixed overhead is acceptable for the program.

6.3 Checkpointing Performance

A typical application scenario of process backtracking is simulation backtracking. Setting checkpoints for the simulation program is the core content of the simulation backtracking, and its efficiency is closely related to the backtracking efficiency. We chose a CISE-based [19] simulation program with a run time of approximately 150s to find the impact of checkpoints on the simulation program execution. The memory checkpoint tool we use is CRIU [22]. We execute the simulation program after instrumentation, then set multiple checkpoints uniformly during its execution, and finally record the execution time of the entire simulation program. The experimental results we recorded are shown in Fig. 7.

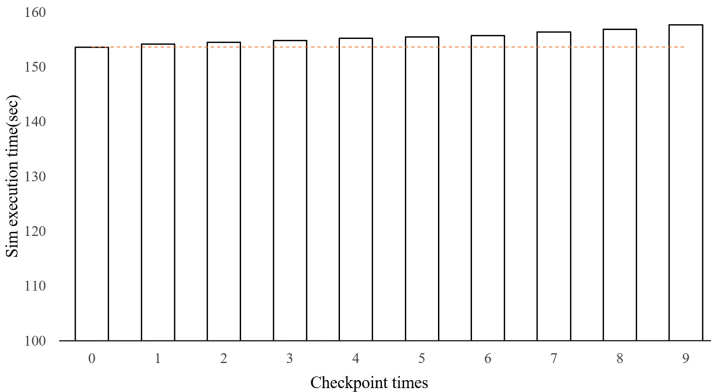


Fig. 7. The performance of the checkpoint setting on the simulation program.

The abscissa in the figure represents the number of checkpoints set on the simulation program. The abscissa is 0 means the time required to execute the simulation program itself. A single checkpoint has a limited impact on the execution time of the simulation program. With the increase in the number of checkpoints, the execution time of the simulation program increases linearly and slowly. For simulation programs that require frequent backtracking, the time overhead of setting checkpoints multiple times is tolerable.

7 Conclusion

We have described a new checkpoint idea, which can not only create process checkpoints, but also help process traceback (this is very useful for simulation programs). The system uses dynamic instrumentation tools to intercept

the native file access interface and insert a virtual file layer to unify the volatile and persistent data of the process without modifying the imitation source code. Although performance is the most important issue of the process, our experimental results on micro-benchmarks and practical applications show that the cost of introducing dynamic instrumentation and virtual file layer is acceptable, and the impact on the process itself is very limited. Our experience shows that the use of dynamic instrumentation tools to insert virtual file layer can satisfy the checkpoint setting requirements of conventional processes, and also provides a solution for process backtracking.

References

1. Wang, Y.M., Huang, Y., Vo, K.-P., Chung, P.-Y., Kintala, C.: Checkpointing and its applications. In: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, p. 22. Institute of Electrical and Electronics Engineers, Inc., Washington, DC (1995)
2. Pei, D.: Modification operations buffering: a low overhead approach to checkpoint user files. In: Proceedings of IEEE 29th Symposium on Fault-Tolerant Computing, Madison USA, pp. 36–38 (1999)
3. Duell, J.: The design and implementation of Berkeley Lab’s Linux checkpoint/restart. Berkeley Lab Technical report, LBNL-54941 (2002)
4. Duell, J., Hargrove, P., Roman, E.: Requirements for Linux checkpoint/restart. Berkeley Lab Technical report, LBNL-49659 (2002)
5. Roman, E.: A survey of checkpoint/restart implementations. Berkeley Lab Technical report, LBNL-54942 (2002)
6. Sankaran, S., et al.: The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. In: LACSI Symposium, LBNL-53808 (2003)
7. Paul H., Duell, J.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters. In: Proceedings of SciDAC 2006, LBNL-60520 (2006)
8. Lyubashevskiy, I., Strumpfen, V.: Fault-tolerant file-I/O for portable checkpointing systems. *J. Supercomput.* **16**, 69–92 (2000)
9. Rashid, R., et al.: Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. Comput.* **37**(8), 896–908 (1998)
10. Zhong, H., Nieh, J.: CRAK: Linux checkpoint/restart as a Kernel module. Technical report CUCS-014-01, Department of Computer Science, Columbia University (2001)
11. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The design and implementation of Zap: a system for migrating computing environments. In: Proceedings of the Fourth Symposium on Operating Systems Design and Implementation. *ACM SIGOPS Operating Systems Review* (2002). <https://doi.org/10.1145/844128.844162>
12. Chung, P.E., Huang, Y., Yajnik, S.: Checkpointing in CosMiC: a user-level process migration environment. In: Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems. IEEE Computer Society (1997)
13. Weihl, W.E.: Transaction-processing techniques. In: *Distributed Systems*, pp. 329–352. ACM Press/Addison-Wesley Publishing, New York (1993)
14. Wang, Y.M., Chung, P.E., Huang, Y.: Integrating checkpointing with transaction processing. In: Proceedings of 27rd Fault-Tolerant Symposium, Seattle, Washington, pp. 24–27. IEEE Computer Society (1997)

15. Ouyang, J., Maheshwari, P.: Supporting cost-effective fault tolerance in distributed message-passing applications with file operations. *J. Supercomput.* **14**, 207–232 (1999)
16. Pei, D., Wang, D., Shen, M., Zheng, M.: Design and implementation of a low-overhead file checkpointing approach. In: *Proceedings of the Fourth International Conference/Exhibition on High Performance Computing, Asia-Pacific Region*, pp. 439–441 (2000)
17. Liu, S., Wang, D., Zhu, J.: A files checkpointing approach based on virtual file operations. *J. Softw.* **13**(8), 1528–1533 (2002)
18. Jeyakumar, A.R.: *Metamori: a library for incremental file checkpointing*. Master’s thesis, Virginia Tech, Blacksburg (2004)
19. Qing, D., et al.: Research of component-based integrated modeling and simulation environment. *J. Syst. Environ.* **04**, 900–904 (2008)
20. Xue, R., Chen, W., Zheng, W.: CprFS: a user-level file system to support consistent file states for checkpoint and restart. In: *Proceedings of the International Conference on Supercomputing*, pp. 114–123 (2008)
21. FUSE Doc. <https://www.kernel.org/doc/html/latest/filesystems/fuse.html>. Accessed 28 Apr 2020
22. CRIU Homepage. <https://criu.org/Main.Page>. Accessed 28 Apr 2020
23. IOzone Homepage. <http://www.iozone.org/>. Accessed 29 Apr 2020
24. BWA Homepage. <https://github.com/lh3/bwa>. Accessed 29 Apr 2020
25. Pin Doc. <https://software.intel.com/sites/landingpage/pintool/docs>. Accessed 29 Apr 2020
26. Agrawal, H., Demillo, A.R., Spafford, H.E.: An execution-backtracking approach to debugging. *IEEE Softw.* **8**(3), 21–26 (1991)
27. Matthews, G., Hood, R., Johnson, S., Leggett, P.: Backtracking and re-execution in the automatic debugging of parallelized programs. In: *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, UK, pp. 150–160 (2002)
28. Agrawal, H., DeMillo, R.A., Spafford, E.H.: Debugging with dynamic slicing and backtracking. *Softw. Pract. Exper.* **23**(6), 589–616 (1993)

Model, Simulation and Evaluation of Architecture



Directory Controller Verification Based on Genetic Algorithm

Li Luo^(✉), Li Zhou, Hailiang Zhou, Quanyou Feng, and Guoteng Pan

College of Computer, National University of Defense Technology, Changsha 410073, China
li_luo@nudt.edu.com

Abstract. Directory protocol is the most widely used implementation cache consistency method in large-scale shared memory multi-core processor which is very complex and difficult to verify. In this paper, we propose a random test generation method based on genetic algorithm to verify directory controller of a type of 64-core processor, analyze the test features to code the symbols of genetic algorithm, and evaluate the merits of the test using the fitness function based on functional coverage. We establish the relationship between coverage and test vector, analyze the relationship between coverage and test stimulus through a genetic algorithm. The experimental results show that compared with the pseudo-random method, the functional coverage rate of this method is increased by nearly 20%–30%, the detection rate of bugs is relatively high, and the verification efficiency and quality are also improved.

Keywords: Directory-based protocol · Directory controller · Functional coverage · Genetic algorithm (GA) · Coverage directed test generation (CDG)

1 Introduction

Cache is a key component of microprocessor. With the development of processor structure to multi-core and many-core, cache consistency protocol is becoming more and more complex. How to ensure the correctness of consistency protocol has always been the focus of industry and academia. With the increase of the number of processor cores, the cache consistency protocol is the most commonly implementation method. Compared with the snoopy protocol, it has good scalability and reliability, and sends consistency requests accurately adopting peer-to-peer mode [1, 2]. Because of the state space explosion of directory consistency protocol, software simulation verification method has been a common verification means. The biggest challenge of software simulation method is how to generate high-quality test stimuli to cover function points. At the same time, the quality and progress of verification are usually measured by coverage. The goal of this paper is to verify the function of a 64-core processor directory controller. There are 1764 protocol message attribute function points, 58 key status registers in the directory controller, and 1822 functional points in total. The key problem of directory controller verification is how to generate effective test stimuli to cover functional points.

The manual test stimuli take exhausting time, and the random test stimuli cover the function points with great blindness. We propose an approach for automatic coverage directed test generation (CDG) based on genetic algorithm, aim at constructing efficient test generators for checking the important behavior and specification of the design under test (DUT), improving the coverage progress rate; and designing stimulus that can reach uncovered tasks (coverage points). We establish the relationship between coverage and test vector, analyze the relationship between coverage and test through a genetic algorithm. The experimental results illustrate the effectiveness of the proposed algorithm in achieving the goals of CDG. Compared with the pseudo-random method, the functional coverage rate of this method is increased by nearly 20%–30%. The discovery rate of bugs is also improved effectively.

The rest of this paper is as follows. Section 2 reviews related work. Section 3 describes target DUT. Section 4 presents the details of the proposed genetic algorithm. Section 5 illustrates the experimental results. Finally, Sect. 6 concludes the paper.

2 Related Work

The software simulation method has always been a common verification method in the chip research. The test stimulus of software simulation mainly includes manual test, random test and coverage directed test generation (CDG). The manual test stimulus takes exhausting time and cost. Sometimes it is difficult to meet the needs of a large number of test vectors covering a wide range in regression.

Nagamani, A.N. [3] presented the first implementation of a generation framework that used feedback from coverage analysis to direct microarchitecture simulation, during the verification of IBM z-series servers, a CDG system had the potential to bring considerable advantages for a reasonable price. For models of a larger scale, the contribution was even greater. These advantages can save machine and personal time, and thus save money overall.

Fine S et al. put forward a random test generation method based on Bayesian network. This method was successfully applied to verify the PowerPC Northstar pipeline. Through improving the parameters of the training network, the verification efficiency and the hit rate of the simulation vector were effectively improved [5, 6].

Ilya Wagner etc. [7] used a random instruction generator driven by a Markov chain model, which has higher error efficiency compared with the random instruction test generation technology.

Yi Jiang-fang [8] used Bayesian network to describe the relation between the inputs and the branch statements. The new simulation vectors were generated by reasoning on the network. Experiments results indicated that the average vector length generated by the Bayesian network using different reference algorithms is about 10% of the original one, but the best path coverage even exceeds the original one.

Ai Yang-yang [9] discuss the Cache coherency protocol, analyze the coverage directed test generation (CDG) method based on Bayesian network reasoning and applied the method to Cache consistency verification. Taking the verification of the Cache coherence protocol of the FT processor as an example, the results show that the CDG method can increase coverage by nearly 30% in comparison with the pseudo-random test.

Bose M. [10] presented a genetic algorithm based framework to automatically generate biases. They targeted utilization of specific buffers for a new version of the PowerPC architecture. The results showed that the GA is effective in achieving high buffer utilization. Also, in targeting multiple objectives, the best approach to be used depends on whether the objectives were related.

Wang Shu-peng [11] proposed a coverage-directed test generation based on genetic algorithm (GA), which was used to verify two high-performance 32-bit multi-core processors. Results show that the proposed method can significantly reduce simulation time and improve verification efficiency.

Shen Hai-hua [12] presented genetic algorithm (GA) based coverage directed test generation (CDG), and built a coverage directed test generation platform. Experimental results showed that CDG can apparently accelerate the verification process and improve the reached coverage from 83.3% to 91.7%, which implied that verification efficiency was greatly improved and skilled manpower was cut down dramatically.

Nagamani, A.N [13] proposed a genetic algorithm based on heuristic test set generation method for fault detection in Reversible Circuits, which avoided the need for an exhaustive search. The approach validated on benchmark circuits considering missing-gate fault (complete and partial), bridging fault and stuck-at fault with optimum coverage and reduced computational efforts.

Genetic algorithms are intelligent approach to automate the generation of effective solution for black-box optimization without requirements of experience knowledge and resources. Only according to the input and output of DUT, Genetic algorithms learn the relationship between the feedback test vector and the coverage point automatically, which improves the automation of verification.

Directory-based protocols are very complex, which need rich test vector test to meet the requirement of functional verification, and random test produces a lot of redundant stimulus. In order to break through the bottleneck of verification, the genetic algorithm does not traverse the whole search space, we use genetic algorithm to mine the relationship between the coverage and the stimulus, and then guide the generation of random test stimulus, improve the growth rate of coverage, reduce the simulation time of redundant stimulus, and increase the efficiency of verification.

3 Background

Our research background is a 64-core processor developed, which is global shared memory and CMP structure, as shown in the Fig. 1, including processor core, cache, network on chip (NOC), directory control unit (DCU) and memory control unit (MCU).

Core: CPU core, which completes the scheduling and execution of instructions.

Cache: Two cores share one cache.

NOC: Interconnection network on chip, which provides information message exchange between caches and between cache and external memory of the CPU.

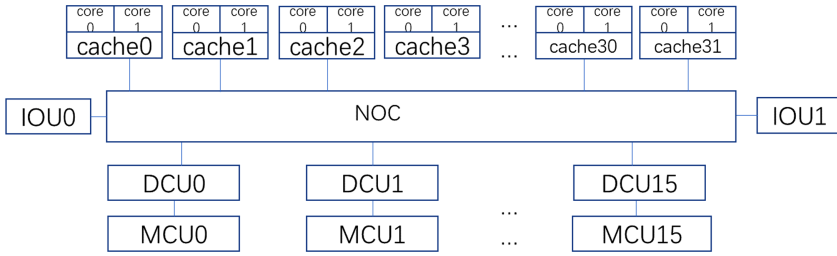


Fig. 1. Target system of 64-core CMP structure

DCU: directory controller, which records the usage of data block copies in each cache, and completes the maintenance of data consistency between each cache.

MCU: memory controller, attaching External DRAM memory, to achieve read-write access control of memory.

IOU: IO controller, connecting the PCIe device controller and other IO devices.

Two cores share a cache, and each cache line data has four states of MESIO (Modified, Exclusive, Shared, Invalid, Owned). The function of the directory controller is to support global data sharing, track and record the use of each cache line data, generate access requests to the memory controller MCU, process cache consistency protocol messages, send snoopy requests, accept cache snoopy responses, and complete cache requests.

The directory controller records the tags of all cache line dates and the using status of data copies in the cache, it tracks and modifies them according to the current received message commands and directory status to maintain the cache consistency among the caches in the whole processor.

The directory controller adopts the distributed directory implementation method. The directory protocol implementation method is divided into centralized and distributed [14], the centralized directory design is simple, the network traffic around the target controller is often the access hotspot, thus affecting the consistent transmission delay, resulting in the network power consumption hotspot. In order to improve the directory parallel processing ability of multi-core processors, the cache consistent transactions of the target system are evenly distributed to the DCU, and a DCU has the same address code for the same memory. In order to further improve the parallel processing performance, the DCU is divided into two individual banks, which are cross accessed by 6th address bit, 16 DCUs are designed on the 64-core chip. The directory table adopts the configurable group association mode, and the test configuration is 24-way 64 entries group association organization. The directory table is shown in the Fig. 2.

The organization and addressing mode of directory entry are the same as that of the cache tag. Before filling the cache line data, DCU allocates a directory entry to record using status of cache line data. If the cache line is replaced or does not retain the data copy, the directory will retrieve the corresponding directory entry (Fig. 3).

DCU entry mainly includes tag, busy, valid, vector, ECC bits and other information. The tag of the DCU entry, that is, the high 24-bit of the memory address. The busy defines the busy status of the directory entry. If the snoopy request is not completed, the new request with the same address cannot be processed. Valid means that the directory

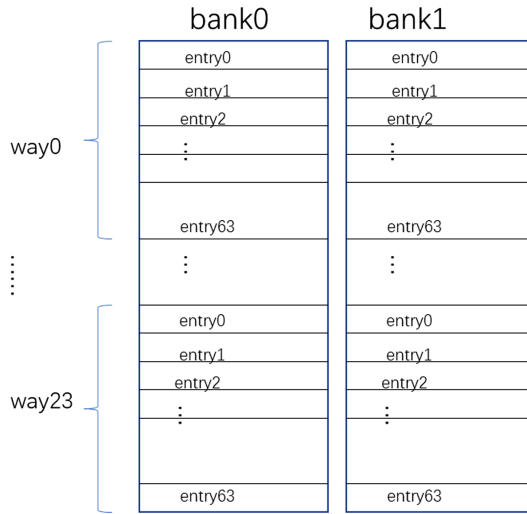


Fig. 2. Directory table structure of DCU

V	B	tag	vector	ECC
---	---	-----	--------	-----

Fig. 3. Content of DCU entry content

entry is valid, at least one cache line has data copy. Vector defines which cache has data copy. ECC check bit is the Hamming check bit of the catalog entry.

The 16 DCUs of the target system are relatively independent, and DCU0 manages the cache consistency record mapping the memory access space of mcu0. Therefore, the functional verification of the directory controller can be tested against an independent DCU0.

A simulation environment is built for dcu0, as shown in Fig. 4. Cache is an IP design, it is replaced by cache model in DCU simulation environment, which is a functional model with an accurate clock, realizes the cache function in the processor. IOU and MCU select the function model with an accurate clock, simulates IO transaction and memory access transaction.

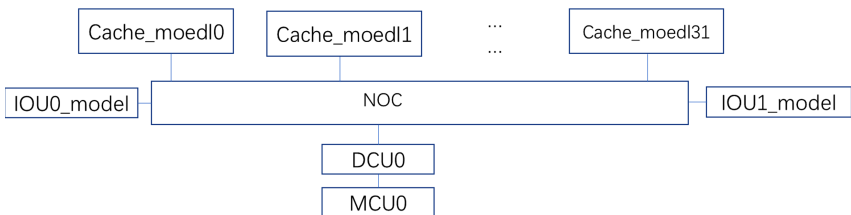


Fig. 4. Simulation environment of DCU0

DCU0 can receive requests from 32 caches or 2 IO controllers, MCU response, cache snoopy response, output cache snoopy request, or memory access or IOU access request. These input and output cache consistency information of dcu0 is transmitted in the format of on-chip network message, which can be divided into four categories: read or write request of cache/IOU, snoopy request, read or write response/snoopy response.

The main verification of DCU0 is the correct processing of cache consistency message and the directory table function, including directory entry hit, replacement, directory entry busy hit, directory table full hit, directory entry misses and other function points. There are 1764 protocol message attribute function points, 58 important status registers, and 1822 functions in total. Random test is used in the simulation process, redundancy test vectors often appear, which is easy to occur the verification period, the coverage of function points increases slowly. The key of DCU verification is how to generate effective test stimulus automatically, in order to break through the bottleneck of coverage and speed up the function verification.

We use a genetic algorithm to mine the relationship between the coverage and the stimulus, and then guide the generation of random test stimulus, improve the growth rate of coverage, reduce the simulation time of redundant stimulus, and increase the efficiency of verification. The effective stimulus generation is abstractly transformed into a genetic algorithm with improving coverage evolution.

4 Test Generation Based on Genetic Algorithm

The main process of genetic algorithm can be described as follows: the possible solution code of the problem is expressed as chromosome, and a chromosome population is randomly generated. Then, the chromosome individuals in the population are placed in a certain environment, and according to the survival principle of the fittest, the individuals with better adaptation environment are selected for replication, crossover, mutation and other operations, the next generation of individuals who are more adaptable to the environment. Such a generation evolves and keeps the offspring with large adaptive function. When the fitness function reaches the threshold, the evolution stops, and the optimal solution is obtained.

The essential feature of genetic algorithm is to code chromosomes through feasible solutions of the problem, to maintain optimization of crossover and mutation operators between generations, to define fitness function fitness in genetic algorithm, to judge the quality of chromosomes in the population, and to achieve multi-directional and global search to find the optimal solution of the problem.

4.1 Question Encoding

When genetic algorithm is used to solve the optimization problem, it is necessary to map the feasible solution of the problem from the solution space to the search space that the genetic algorithm can deal with, that is, to code the feasible solution of the problem with chromosome code.

The test generator is based on a genetic algorithm, its genetic code is related to the test stimulus. The genetic code is transformed into a feature vector, i.e. gene, by extracting the features of test stimulus. The gene sequence is built to obtain a chromosome.

Test stimulus features mainly come from the following aspects: first, opcode, non-cacheable read, shared read, noncacheable write, and cache maintenance commands of the message, etc.; second, the number of send SENDID, which indicates which cache and IO controller the data comes from, the third is address correlation of each message contained, if it is relevant, then the read and write addresses are the same. The fourth is bank number, the fifth is length encoding, whether the message request is 1 byte, 2 bytes, 16 bytes or 64 bytes data. So, gene expression includes five feature vectors.

In the implementation of genetic algorithm, genetic coding adopts symbol coding, and each bit field represents a feature vector. Using symbol coding corresponds to the problem itself, which is simple, easy to understand, and faster and more stable than binary coding in solving optimization problems. For example, gene expression is shown in Fig. 5. For gene coding (4,5,1,1,0), it can determine that cache4 sends a request that the opcode is op5. In this chromosome, the address is the same as the first gene address, access the directory table bank1 and read 1 byte of data.

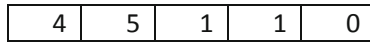


Fig. 5. An example of gene

A test stimulus consists of multiple test vectors, i.e. forming a chromosome. We define 32 genes to form a chromosome. As shown in Fig. 6, the chromosome is composed of 32 genes, Gene 0 is cache4, which sends out a request that the opcode is op5. The address is at the first address of this chromosome address, that is, random address. Access directory bank1, and read 1 byte of data. Gene 1 is cache0 which sent out a request with opcode OP1, the address is the same as the first gene address of this chromosome, access the directory table bank1, and read 64 bytes of data; gene 31 is the request of IOU 2 that the opcode is op0, the address is different from the first gene address of this chromosome, and read 16 bytes of data.

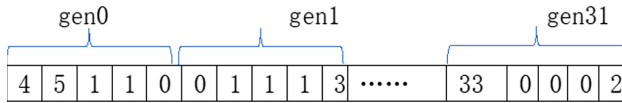


Fig. 6. An example of chromosome

4.2 Fitness Function

Genetic algorithms use a fitness value to evaluate the quality of chromosome. The evaluation of solutions represented by fitness values is important to guide the learning and evolution process in terms of speed and efficiency. The function verification based on simulation is to get the maximum function coverage in a short period of time. Different scenarios and functions defined by test stimulus are reflected by the population of chromosomes. Fitness function in genetic algorithms is used to judge the quality of

chromosomes in the population. Therefore, in the application of this study, the fitness function is to reach the most function points. In genetic algorithms, a chromosome C represents a test stimulus, and there are N function points to be tested and covered in the DCU0. For a chromosome C, its fitness function fitness. Its fitness function is the number of function points of coverage. The number of function points included in chromosome C is M, the gene i of chromosome C covering the jth function point is defined as cov[i, j], and Covering the jth function point of chromosome C is defined as follows:

$$\begin{cases} \text{cov}[j] = |(\text{cov}[i, j])|, \text{ for, } i = 1, 2, \dots, M \\ \text{s.t. cov}[i, j] = 1, \text{ Gene } i \text{ covers the } j\text{th function;} \\ \text{else cov}[i, j] = 0, \text{ Gene } i \text{ does not cover the } j\text{th function} \end{cases} \quad (1)$$

$$\text{Fitness}_c = \max(\sum_{j=1}^N (w_j * \text{cov}[j]))/N \quad (2)$$

w_j represents the weight and importance of the jth function point. Its value can be adjusted according to the design features. For example, when only the module of the directory table bank0 is tested, the function point weight of bank1 can be set to 0.

4.3 Mutation Operator and Crossover Operator

The mutation of genetic algorithm itself is a kind of local random search, which is combined with random and crossover operators to ensure the effectiveness of genetic algorithm, make genetic algorithm have the ability of local random search, and keep the diversity of population, so as to prevent premature convergence. In order to avoid invalid operation, we adopt fixed-point mutation. The position of mutation operation is the first and second position of gene, as shown in Fig. 7, an example of mutation operation.

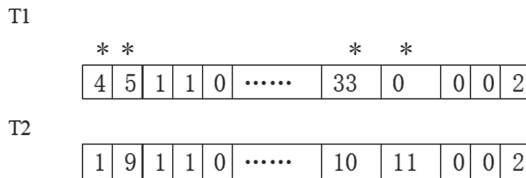


Fig. 7. Example of mutation operation

An example of mutation operation shown in Fig. 7 is to perform the mutation operation on chromosome T1. If there is a mark “*” in the characteristic position, the mutation will occur. T1 generates a new chromosome T2 through mutation. We select the mutation feature as opcode and SENDID. We need to pay attention to the legitimacy of the variation, such as the legality of opcode. IO devices can only send non-cacheable read and non-cacheable write requests, such as sending cacheable requests, this must be an illegal operation and needs to be deleted.

The crossover of genetic algorithm is the operation of replacing and recombining part of the structure from the parent to generate new individuals. Its purpose is to generate new individuals in the next generation, just like the process of human evolution, so

that the search ability of genetic algorithm can be improved greatly. Crossover is carried out according to probability. The higher the frequency of crossover, the faster the optimal solution can converge, but too high will lead to premature convergence. Common crossover includes single point crossover, multi-point crossover, uniform crossover, etc. we select single point crossover, and the crossover operator will randomly exchange some feature bits of two chromosomes according to the crossover rate, so as to generate a new feature combination. The purpose of crossover is to combine the useful features together to produce more effective and active coverage of function points. The specific operation is to set a crossing point in the chromosome code, then exchange the partial structure of the two chromosome codes before and after the crossing point, and form two new chromosome codes, that is, two new test stimuluses. Figure 8 introduces an example of a single point crossing, through which two chromosomes T1 and T2 in Fig. 9 can be changed into two new chromosomes T'1 and T'2.

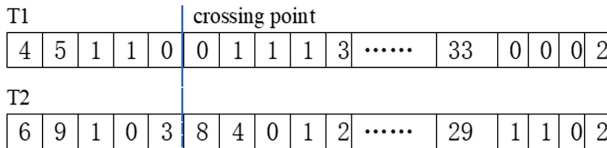


Fig. 8. Chromosomes before cross operation

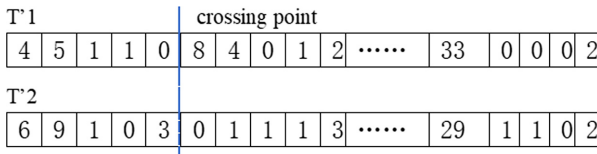


Fig. 9. New chromosomes after cross operation

4.4 Parameters of Genetic Algorithm

The genetic algorithm needs to determine the size of the test set, that is, the number of genetic populations POPSIZE. When the POPSIZE value is small, the calculation time of the algorithm is short, but the probability of the algorithm converging to the optimal solution is low, that is, the global search ability is small, and the local optimal solution may be obtained instead of the global optimal solution. With the increase of POPSIZE, the probability of convergence to the optimal solution will increase, but the calculation time of the algorithm will also increase significantly. Our algorithm defines the population size as 32 chromosomes.

Genetic algorithm crossover is built on probability. The higher the crossover frequency is, the faster the optimal solution can converge, but too high will lead to premature convergence. Mutation is kind of local random search, at the same time, it makes the genetic algorithm keep the diversity of the population, so as to prevent the premature

convergence. In the mutation operation, the mutation rate cannot be too large. Otherwise it may degenerate into random search. At this time, some important mathematical characteristics and search ability of the genetic algorithm no longer exist. In a compromise, the crossover probability $PC = 0.8$ and the mutation probability $PM = 0.1$ are defined here.

5 Experimental Results

The parameters of the genetic algorithm are set as follows: population size $POPSIZE = 32$, Maximum number of evolutionary iterations $MAXGENERATION = 10$, crossover probability $PC = 0.8$, mutation probability $PM = 0.1$. According to fitness function of Formula-1 and Formula-2, chromosome evaluation in genetic algorithm is realized, The simulation scenario of testbench1 is 32 caches and two IOU devices accessing a directory controller DCU0, sending out 4 K requests, simulating 90 K cycles, obtaining 100% coverage, pseudo-random test stimulus reaches 73%, as shown in Fig. 10, testbench2 simulation scenario 2 is 4 caches and two IOU devices accessing DCU0, sending out 256 requests, simulating 1 K cycles. At the same time, the pseudo-random test motivation reaches 81% of the functional coverage, as shown in Fig. 11.

In the regression test of directory simulation, there are 24 bugs found by testbench1 of genetic algorithm, 5 bugs of high quality, and 21 bugs found by testbench2 of genetic algorithm. The results and performance comparison of the algorithm are shown in Table 1.

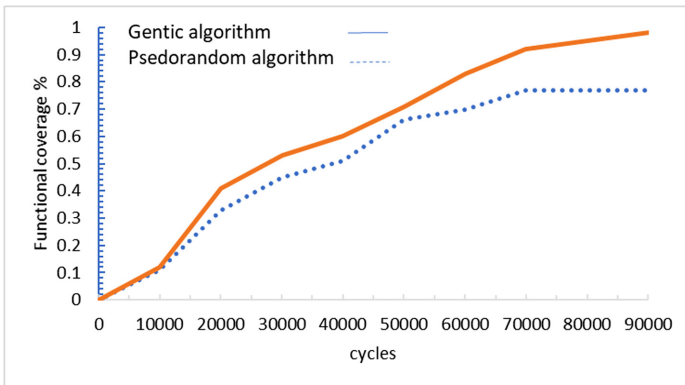


Fig. 10. A coverage comparison of testbench1 simulation scenario

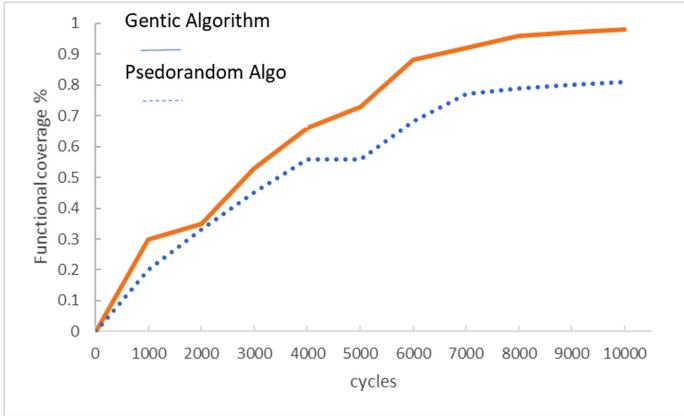


Fig. 11. B coverage comparison of testbench2 simulation scenario

Table 1. Performance comparison of algorithms

Test scenario	Algorithm	Simulation time (cycles)	Test vector	Functional coverage	Detecting bugs in regression test
TestBench1	Genetic algorithm	90k	4k	100%	24
TestBench1	Pseudorandom algorithm	90k	4k	73%	19
TestBench2	Genetic algorithm	10k	1k	100%	21
TestBench2	Pseudorandom algorithm	10k	1k	81%	15

6 Conclusions

In this paper, we propose a random test generation method based on genetic algorithm to verify directory controller of a type of 64-core processor, which uses a fitness function based on function coverage to evaluate the quality and value of verification. The genetic algorithm is used to establish the relationship between the coverage analysis results and the effective stimulus to direct the generation of higher quality tests. The experimental results demonstrate that compared with the pseudorandom test generator, the proposed test generator can achieve higher function coverage in a short time, reduce the verification time and improve the verification efficiency. The parameters of the genetic algorithm in this paper are fixed, which is likely to cause the genetic algorithm to fall into the local optimal solution. In the future, we will do further research on the adaption of the parameters and test weight of the genetic algorithm, further expand the local optimal solution space of the genetic algorithm, and generate higher quality tests.

Acknowledgments. This work is supported by National Key Research and Development Program of China No. 2018YFB0204301.

References

1. Hill, M.D., Sorin, D.J., Wood, D.A.: A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures on Computer Architecture, November 2011
2. Simoni, R., Horowitz, M.: Modeling the performance of limited pointers directories for cache coherence. In: Proceedings of the 18th International Symposium on Computer Architecture, pp. 309–318 (1991)
3. Nativ, G., Mittennaier, S., Ur, S., Ziv, A.: Cost evaluation of coverage directed test generation for the IBM mainframe. In: Proceeding of the 2001 International Test Conference, Baltimore, pp. 793–802 (2001)
4. Fine, S., Ziv, A.: Coverage directed test generation for functional verification using Bayesian networks. In: Design Automation Conference, pp. 286–291 (2003)
5. Braun, M., Fine, S., Ziv, A.: Enhancing the efficient of Bayesian network based coverage directed test generation. In: Proceedings of IEEE International High-Level Design and Test Workshop, Sonoma, pp. 75–80 (2004)
6. Fine, S., Freund, A., Jaeger, I., Naveh, Y., Mansour, Y.: Harnessing machine learning to improve the success rate of stimuli generation. *IEEE Trans. Comput.* **55**(11), 1344–1355 (2006)
7. Wagner, I., Bertacco, V., Austin, T.: Microprocessor verification via feedback-adjusted Markov models. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **26**(6), 1126–1138 (2007)
8. Yi, J., Dong, T., Xu, C.: An efficient approach to simulation vector generation using Bayesian network. *J. Comput.-Aided Des. Comput. Graph.* **19**(5), 616–621 (2007). (in Chinese)
9. Ai, Y., Luo, L., et al.: A Bayesian network based test generation method for cache coherency protocol verification. *Comput. Eng. Sci.* **39**(8), 1397–1402 (2017). (in Chinese)
10. Bose, M., Shin, J., Rudnick, E.M., et al.: A genetic approach to automatic bias generation for based random instruction generation. In: Proceedings of Congress on Evolutionary Computation, Seoul, pp. 442–448 (2001)
11. Wang, S., Huang, K., Yan, X.: Coverage directed test generation based on genetic algorithm. *J. Zhejiang Univ. (Eng. Sci.)* **50**(3), 581–588 (2016)
12. Shen, H., Wang, P., et al.: A coverage directed test generation platform for microprocessors using genetic approach. *J. Comput. Res. Dev.* **46**(10), 1612–1625 (2009)
13. Nagamani, A.N., et al.: A genetic algorithm-based heuristic method for test set generation in reversible circuits. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* **37**(2), 324–335 (2018)
14. Ros, A., Acacio, M.E., García, J.M.: A scalable organization for distributed directories. *J. Syst. Archit.* **56**(2–3), 77–87 (2010)



Prediction and Analysis Model of Telecom Customer Churn Based on Missing Data

Rui Zeng^(✉), Lingyun Yuan, Zhixia Ye, and Jinyan Cai

School of Information Science and Technology, Yunnan Normal University, Kunming, China
zengruyn@126.com, blues520@sina.com, yezxleaf@aliyun.com,
jinyanninhao@126.com

Abstract. In the field of business data analysis, customer churn prediction analysis plays an important role. This paper combines traditional statistical prediction methods and artificial intelligence prediction methods to propose a customer churn prediction analysis model based on missing data in an attempt to explore a new solution in this field. Based on the missing data in this model, factor analysis method and data mining technique are used to generate key factor sets and their values to form input neurons and their initial values. The number of hidden layer neurons was determined by combinatorial prediction. Using the improved genetic algorithm, the initial weight and threshold of BP network are determined. Finally, the prediction results and key attribute data related to the prediction results are generated for decision makers to analyze the problem. The experiment evaluates the model from the aspects of accuracy, precision, recall, and f-measure, which proves that the model is effective.

Keywords: Factor analysis · Data mining · Genetic algorithm · Predictive analysis

1 Introduction

In the increasingly competitive environment, the cost of retaining old customers is 20% - 10% of developing new customers. Compared with new customers, loyal old customers can bring more benefits to enterprises. Therefore, it is very important for telecom companies to adopt a defensive marketing strategy. Customer churn analysis and prediction play an important role in defensive marketing strategies. The problem of customer churn is a two classification problem based on the limited characteristics of customers. With the research on this issue, the intelligence of building models has gradually deepened. The forecasting methods mainly include traditional statistical forecasting methods, artificial intelligence forecasting methods, and statistical theoretical forecasting methods. The algorithms used in traditional statistical prediction methods include decision trees [1], logistic regression [2], Bayesian classifiers [3], and clustering [4]. The papers [5, 6], and [7] respectively build prediction models based on traditional statistical prediction methods. Decision tree is a traditional statistical prediction method with better effect. The

advantage of decision tree is simple and fast, especially suitable for large-scale data processing. However, its algorithm has poor anti-noise performance, and its performance is not good when processing data with strong feature correlation. Statistical theory prediction methods mainly involve support vector machine. SVM has many unique advantages in solving nonlinear and high-dimensional pattern recognition problems, but it is difficult to implement for large-scale training samples. Artificial intelligence prediction methods mainly include artificial neural networks [8], evolutionary learning algorithms [9], and self-organizing maps [10]. Article [11] discussed the best method of ANN in predictive analysis and established a predictive model. Neural network has good coordination adaptability, distributed storage and great fault-tolerant performance, which makes it play an increasingly important role in the field of data analysis. How to realize analysis and prediction simply and effectively is the focus of this modeling work. BP neural network is a simple and effective neural network, which can effectively make up for the shortcomings of decision tree and support vector machine. But BP neural network has its limitations. Therefore, this model intends to improve BP neural network to realize the analysis and prediction of customer churn in a simple and effective way.

Missing data is an important factor that affects the prediction effect. This model uses factor analysis and improved clustering analysis to effectively solve the problem of missing data's impact on prediction results. Based on factor analysis, improved clustering analysis and improved BP neural network, this paper proposes a customer churn prediction analysis model based on missing data in an attempt to explore a new solution in this field. Experiments based on historical telecom operation data prove that this solution is indeed effective.

2 Model Implementation

This model consists of four parts: the input neuron determination module, hidden layer neuron determination module, the initial weight and threshold determination module, and the analysis and prediction module, as shown in Fig. 1.

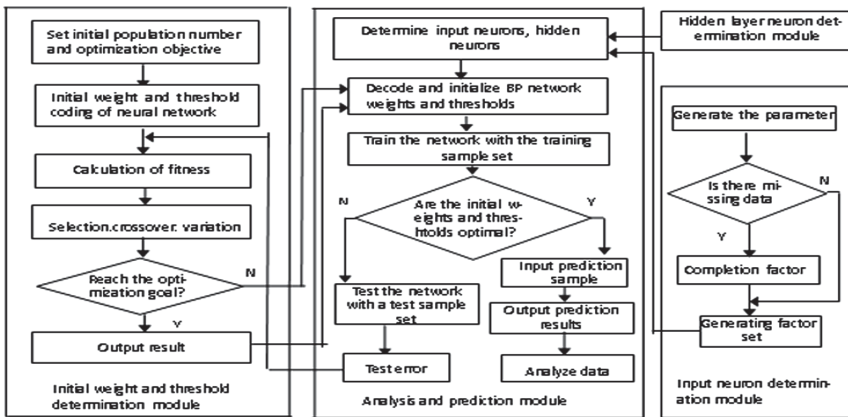


Fig. 1. System model and flow chart

The input neuron determination module uses the factor analysis method and data mining technology to generate a set of factors and their values based on the missing data to form the input neuron and its initial value. The hidden layer neuron determination module uses the combined prediction method to determine the number of hidden neurons. The initial weight and threshold determination module uses an improved genetic algorithm to determine the initial weight and threshold of the BP network. Input the result data of the three modules into the analysis and prediction module, use BP network to generate prediction results and key attribute data related to the prediction results for decision makers to analyze the problem.

2.1 Input Neuron Determination Module

The input neuron determination module first determines the module parameters. Then determine whether there is missing data. If there is missing data, factors based on missing data will be completed using factor analysis and improved clustering algorithms. If the data is complete, factors based on complete data will be generated using factor analysis. All the factors generated by this module constitute a factor set based on the problem of churn prediction analysis. The module flow chart is shown in Fig. 1.

2.1.1 Generation of Module Parameters

The work of this module is divided into two parts. In the first part, the model parameters are obtained based on the factor analysis method, including factor loading matrix, special factor matrix, high-load attributes constituting factor F_i and weight β_{ij} . In the second part, we use the improved clustering algorithm to obtain the model parameters: each factor cluster set.

2.1.1.1. Factor Load Matrix, Special Factor Matrix

Factor analysis is a statistical technique for extracting common factors from attribute groups. The basic purpose is to group several closely related attributes into the same class, and each class becomes a factor. Use a few factors to reflect most of the information of the attribute class. Each attribute can be expressed as a linear function of factors and a specific factor, and expressed as a matrix:

$$x = AF + \varepsilon \quad (1)$$

In the above $A = a_{ij}$ is the factor loading matrix, $F = F_i (i = 1, \dots, n)$ is the factor matrix, and, $\varepsilon = \varepsilon_i (i = 1, \dots, n)$ is the special factor matrix. In this module, the complete data in the data set is taken as the data sample for factor analysis to obtain the factor load matrix A and the special factor matrix.

2.1.1.2. High Load Attribute and its Weight

When the load matrix is rotated, only a few attributes of each factor have high load, and the load of other attributes is small. Therefore, the factors can be expressed as a linear combination of high-load attributes, that is, a factor score function, such as formula 2.

$$F_i = \beta_{i1}X_1 + \dots + \beta_{ij}X_j (i = 1, 2, \dots, m) \quad (2)$$

The high load attribute X_j and its weight β_{ij} constituting the factor F_i can be obtained from the above formula.

2.1.1.3. Factor Cluster Set

Clustering analysis is to classify data samples according to the similarity of features, so that the data of the same group are as similar as possible, and the data of different groups are as different as possible. This module improves the k-means algorithm to obtain the clustering set of factors.

K-means algorithm is one of the classic algorithms in partitioning methods. Based on the provided features, the algorithm iteratively allocates data with high similarity measures into k clusters. The similarity measure of this algorithm uses Euclidean distance. Euclidean distance refers to the real distance between two points in m-dimensional space. In the definition of Euclidean distance, each attribute contributes equally to Euclidean distance, so Euclidean distance is not satisfactory in practical applications. In this module, the Euclidean distance is improved by giving different weights to each attribute to produce Euclidean distance which reflects the characteristics of different factors. The rationality of weight setting is a problem that must be solved. Factor analysis can solve this problem perfectly. The attributes in the improved Euclidean distance are composed of high-load attributes $x_i (i = 1, \dots, n)$ in the factor, and the weights are the weights $\beta_i (i = 1, \dots, n)$ of the high-load attributes x_i . The formula is as follows:

$$d(x, y) = \sqrt{\beta_1(x_1 - y_1)^2 + \beta_2(x_2 - y_2)^2 + \dots + \beta_n(x_n - y_n)^2} \quad (3)$$

The above formula $x_i (i = 1, \dots, n)$ and $y_i (i = 1, \dots, n)$ are the two sample points of the i th high-load attribute of the factor $F_j (j = 1, \dots, m)$ and β_i is the weight of the i th high-load attribute.

There are two problems in the application of K-means algorithm. First, it is very sensitive to the selection of the initial point, which may lead to the k-means algorithm converging to the local optimum. Second, divide the data into k clusters, and how to determine the value of k. The improved bisecting k-means algorithm does not need to pre-determine k value and solve local optimal problem.

In this algorithm, SSE (sum of the squared errors) is used to measure the quality of clustering. SSE is the clustering error of all samples. With the increase of the number of clusters k, the sample division will be more refined, the degree of aggregation of each cluster will gradually increase, and SSE will gradually decrease. When k is less than the real number of clusters, the SSE decreases greatly due to the increase of k, which will greatly increase the degree of aggregation of each group. However, when k reaches the real number of clusters, the decrease of SSE tends to be smooth with the increase of k value.

The improved bisecting k-means algorithm is as follows:

Initialize all data into a cluster and divide the cluster into two clusters.

Repeat

- a) Calculate the *SSE* of each cluster.

The formula of *SSE* is as follows, where SSE_j represents the *SSE* of the j th cluster, x_i represents the i th high-load attribute of the factor, β_i represents the weight of x_i , and x^* represents the average of all points in the cluster.

$$SSE_j = \sum_{i=1}^n \beta_i (x_i - x^*) \tag{4}$$

- b) For all clusters with $SSE > \&$, find the cluster with the largest *SSE*. $\&$ represents the threshold of the *SSE*. As the number of clusters k continues to increase, the *SSE* value when the *SSE* becomes smoothly is the threshold.
- c) For the cluster with the largest *SSE* value, arbitrarily select two sample points as the initial centers of the two new clusters, and the cluster was divided into two clusters.

Repeat

- i. According to the average value of sample points in the two newly divided clusters, the sample points are divided into new clusters. Improved Euclidean distance is used to calculate the distance, as shown in formula 3.
- ii. Update the average value of the sample points in each cluster.
Until two clusters no longer changes

Until all $SSE \leq \&$, stop dividing.

2.1.2 Factor Completion

In this model, the missing factor refers to the attribute of the constituent factor including the missing attribute. Formula 3 is used to calculate the distance from the missing factor to the cluster of factor cluster set. The cluster with the shortest distance is the one with this missing factor. The average value of the cluster is given to the missing factor to complete the missing factor.

2.1.3 Acquisition of Key Factor Sets and Their Values

The high-load attributes and their weights are generated based on the input intact attribute data. Factors based on complete data will be generated. The factors generated by the complete data and the factors generated by the missing data constitute the factor set.

2.2 Hidden Layer Neuron Determination Module

There is no positive correlation between the increasing number of hidden layers of BP neural network and the improvement of network performance. Based on the two-class classification problem of customer churn analysis, this module uses a three-layer BP neural network with stable performance for modeling.

In the BP neural network, the choice of the number of hidden neurons is very important. It not only greatly affects the performance of the established neural network model, but also is the direct cause of overfitting. At present, there is no scientific and universal method to determine the number of hidden neurons. If the number of neurons is too large, not only the training time will be increased, but also the irregular content in the sample will be taken into account, which will cause the problem of overfitting and the generalization ability of the network will be reduced.

In order to solve the above problems, scholars at home and abroad have proposed a variety of methods to determine the number of hidden layer units, mainly including trial and error method, empirical formula method, growth method, genetic algorithm and three-point search and other comprehensive optimization methods [12]. The above methods have their limitations. Trial and error method tests by continuously selecting the number of network neurons. Its computing cost is very large and its computing efficiency is low. The growth method starts from a minimum neural network structure, and increases the number of neurons in the hidden layer until it gets a satisfactory number. However, there is no unified solution to the problem of how to terminate the growth. Genetic algorithm, due to its inherent poor climbing ability and slow convergence speed, will cause the search results in the flat area to fall into the local minimum. The three-point search may miss the whole optimal solution. Empirical formula method, whose formula is from the experience of projects and experiments, can only be effective for specific data sets, and cannot be used as a general method for determining the number of hidden neurons. Empirical formula is the simplest and has a certain effect for specific application scenarios.

The study found that each empirical formula has advantages and disadvantages, and it is difficult to achieve the best performance in predictive analysis. Combined prediction is to synthesize the advantages of each single algorithm and make the original sequence comprehensive prediction through different prediction algorithms. In this model, the number of neurons in the hidden layer is determined by the weighted combination of effective empirical formula that verified by experiments. Based on the application scenario of this model, the specific formula is as follows

$$N_{hid} = \beta \cdot \left(\sqrt[2]{N_{out} + N_{in} + a} \right) + (1 - \beta) \left(N_{train} / \delta \times (N_{in} + N_{out}) \right) \quad (5)$$

In the above formula, $a(1 \leq a \leq 10)$ and $\delta(2 \leq \delta \leq 10)$ are constants, $\beta(0 < \beta < 1)$ expressed as weight.

The key issue in the combination forecast is the determination of the weight. The determination of the weight of the combination forecast in this model uses the following methods:

Let the variance of the training errors of the two prediction algorithms is σ_1, σ_2 . The error of the combined prediction model is as follows:

$$e = \beta\sigma_1 + 1 - \beta\sigma_2 \quad (6)$$

Because the two prediction algorithms are independent of each other, the variance of the combined prediction model error is:

$$\sigma(e) = \beta^2\sigma_1 + (1 - \beta)^2\sigma_2 \quad (7)$$

Introducing Lagrange multiplier to find the minimum value of $\sigma(e)$ to get formula 8

$$\beta = \left(\sigma_1 \sum_{i=1}^2 \sigma_i^{-1} \right)^{-1}, \quad \sigma(e) = \left(\sum_{i=1}^2 \sigma_i^{-1} \right)^{-1} \quad (8)$$

Formula 9 for calculating the weight of the combination algorithm of this module can be obtained from formula 8.

$$\beta = \sigma_1^{-1} \sigma(e) \quad (9)$$

2.3 The Initial Weight and Threshold Determination Module

Genetic algorithm (GA for short) is a heuristic algorithm based on the free selection of biological genetics and genetic theory, which simulates the evolution of natural organisms. It seeks the global optimal solution by combining the new individuals generated by mutation and exchange in the population and natural rules of survival of the fittest [12]. Genetic algorithm is mainly composed of the following elements: 1) population individual coding. Encoding forms mainly include binary coding, real coding, floating-point coding, etc.; 2) selecting the appropriate fitness function; 3) genetic operation (selection, crossover and variation). 4) operating parameters. The module algorithm flowchart is shown in Fig. 1.

2.3.1 Population Initialization

In this model, the individual is in the form of binary code, which is composed of four parts: the connection weight between input layer and hidden layer, the connection weight between output layer and hidden layer, hidden layer threshold and output layer threshold. The individual in the population can be expressed as W_i , Where $W_i = [w_1, w_2, \dots, w_k]$, $w_j \in [0, 1]$, k is determined by formula 10.

$$k = n_1 \times n_2 + n_2 \times n_3 + n_2 + n_3 \quad (10)$$

In the above formula, n_1 , n_2 , n_3 is the number of nodes in the input layer, hidden layer and output layer of the neural network.

2.3.2 Fitness Function

In genetic algorithm, fitness function is used to measure the excellence degree of each individual in the population close to the optimal solution. This model uses the error between the actual output and the expected output of the predictive analysis module to construct the fitness function, as shown in formula 11

$$f_i = \frac{1}{\sum_{j=1}^n |y_j - o_j|} + \varepsilon \quad (11)$$

In the above formula, f_i represents the fitness of the i th individual, n is the upper bound of the training set, y_j expected output of the j th test of the BP neural network, and o_j is the actual output of the j th test. ε is a small number, so that the worst individuals in the population still have a chance to reproduce and increase the diversity of the population.

2.3.3 Genetic Manipulation

Genetic operation mainly includes selection, crossover and variation. In the genetic operation, we mainly improve the selection operation.

Genetic algorithms needs different selection pressures at different stages in the evolution of the population. In the early stages, the selection pressure is small to maintain the diversity of the population. In the later stages, the selection pressure is large, so it is necessary to quickly search the optimal solution in a small range. The selection probability of the roulette method has the same selection pressure at different stages in the evolution of the group. In order to dynamically adjust the selection pressure during the evolution of the group, this model uses Boltzman's idea to construct the selection probability p_i of the individual i in the roulette method as follows:

$$p_i = \frac{e^{f_i/T}}{\sum_{i=1}^M e^{f_i/T}} \quad (12)$$

In the above formula, M is the population size, f_i is the fitness of the i th individual, T is the temperature, $T = T_0(0.99^{g-1})$, g is the number of iterations of genetic algorithm. T_0 is the initial temperature, $T_0 = 10(f_{max} - f_{min})$, f_{max} is the maximum fitness, f_{min} is the minimum fitness.

2.3.4 The Operation Parameters

In this module, the operation parameters are mainly improved for the cross probability p_c and the variation probability p_m .

When the crossing probability p_c is large, the new individuals will be generated faster, which will lead to the destruction of high fitness individuals. When the value of p_c is small, the speed of generating new individuals will slow down, which will cause the search process of the algorithm to slow down. It is not easy to produce new individuals when the value of p_m is small. When the value of p_m is large, the genetic algorithm will become a search algorithm. Srinivas proposed an adaptive genetic algorithm. When the individual fitness in the population tends to be the same, the value of p_c and p_m will increase. When the individual fitness in the population is relatively scattered, the value of p_c and p_m will decrease. However, this calculation formula does not consider the value range of p_c and p_m . In this model, the adaptive genetic algorithm is improved by limiting the value range of p_c and p_m . According to the experience, $p_c = 0.6 \sim 0.99$, $p_m = 0.005 \sim 0.01$. The formulas for p_c and p_m for are as follows:

$$P_c = \begin{cases} 0.8 \leq \frac{k_1(f_{max}-f)}{f_{max}-f_{avg}} < 0.99, & f \geq f_{avg} \\ 0.6 \leq k_2 < 0.8 & f < f_{avg} \end{cases} \quad (13)$$

$$P_m = \begin{cases} 0.0075 \leq \frac{k_3(f_{max}-f')}{f_{max}-f_{avg}} < 0.01, & f' \geq f_{avg} \\ 0.005 \leq k_4 < 0.0075 & f' < f_{avg} \end{cases} \quad (14)$$

In the above formula, f_{max} represents the maximum fitness in the population, f_{avg} represents the average fitness in the population, f represents the maximum fitness in two cross individuals, and f' represents the fitness in the variation individuals. k_1, k_2, k_3 and k_4 are constants.

2.4 Analysis and Prediction Module

The analysis and prediction module realize three functions: 1) BP neural network is used to assist the improved genetic algorithm to obtain the optimal initial weight and threshold. 2) BP neural network is used to predict the input data. 3) According to the prediction results, the customer analysis is realized by using the high load attribute of the constituent factors.

BP network (back propagation), which was proposed by a team of scientists led by Rumelhart and McClelland in 1986, is a multilayer feedforward network with error back propagation. It has the advantages of strong nonlinear mapping ability, but the uncertainty of the number of neurons in the hidden layer and the random generation of initial weights and thresholds are all the factors that restrict it to obtain better solutions. The improvement of BP neural network are as follows: determining the number of input neurons and their values, determining the number of hidden neurons and obtaining the optimal initial weights and thresholds. The flow chart of this module is shown in Fig. 1, and the specific algorithm is as follows:

Let n denote the number of neurons in the input layer and h denote the number of neurons in the hidden layer. w_{ij} represents the weights of the i th input neuron to the j th hidden layer neuron. w_j represents the weight of the j th hidden layer neuron to the output neuron. $a_j (j = 1, 2, \dots, h)$ represents the threshold value of the j th hidden layer neuron, b is the output threshold. T_k represents the expected output of k test samples, y_k represents the output of k test samples through BP network.

- ① Obtain h from the hidden neuron determination module.
- ② Get n from the input neuron determination module.
- ③ The initial a_j , b , w_{ij} and w_j are obtained from the initial weight and threshold determination module, ($j = 1, 2, \dots, h, i = 1, 2, \dots, n$).
- ④ Input the training sample set to train the BP network.
- ⑤ Judge if the initial values of a , b , w_{ij} and w_j are optimal. If yes, go to step ⑦.
- ⑥ Input the sample of test set to the trained BP network to calculate $\sum_k^m |T_k - y_k|$, and input the calculation results into the initial weight and threshold determination module to assist calculation the fitness, and go to step ③.
- ⑦ Input the prediction sample to the trained BP network to obtain the prediction value.
- ⑧ The formula 2 is used to obtain the high-load attributes of the prediction samples. The high-load attributes and prediction results constitute the analysis and prediction data.

3 Experimental Verification

In the two classification problem, the prediction accuracy cannot be the only criterion of the model performance. So in the experiment, we use precision, recall and f-measuring to evaluate the model. Recall refers to the proportion of correctly predicted churn customers in the real churn customers. Precision indicates the proportion of correctly predicted churn customers among the predicted churn customers.

The recall and precision are usually contradictory, and a high recall sometimes lead to the decrease of precision. In the evaluation of the model, the f-measure index is used to maximize the precision while controlling the recall. As shown in the formula 15.

$$f\text{-measure} = \frac{(\beta^2 + 1) \times P \times R}{\beta^2 p + R} \quad (15)$$

In the above formula, P represents the precision and R represents the recall. When $\beta = 1$, $f\text{-measure}$ is common $F1$. When the value of $F1$ is larger, the performance of the prediction model is higher.

Seven models were established in this experiment. They are the models proposed in this paper (referred to as M1), decision tree model (referred to as M2), SVM model (referred to as M3), BP neural network without improvement (referred to as M4), M1 model of random initial threshold and weight (referred to as M5), M1 model for obtaining the number of neurons in the hidden layer by a single test formula (referred to as M6) and M1 model of raw data as input neuron (referred to as M7).

The sample set of this experiment involves 60000 pieces of complete sample data of a telecom enterprise from January 2018 to January 2019, which randomly makes the customer's consumption information missing at a proportion of 20%. The sample data set is divided into two parts, one is training set for training model, and the other is for testing model. The training sample data were randomly selected from the training sample set, and seven models were trained with the same sample data. Then 7000 sample data are randomly selected from the test sample set, and seven models are input with the same sample data to obtain the prediction results. The above experiments were carried out 6 times in total. Finally, the average values of the accuracy, recall, precision and f-measurement are used to evaluate the model. The experimental results of M1, M2, M3 and M4 are shown in Table 1.

Table 1. Summary of evaluation indexes of five models

Model	Average number of iterations	Average accuracy rate	Average recall	Average precision	Average f-measure
M1	588.45	86.56%	83.23%	85.43%	0.8422
M2		71.32%	68.32%	73.01%	0.7059
M3		68.76%	71.04%	70.19%	0.7061
M4	701.48	73.67%	72.32%	73.34%	0.7288

According to Table 1, the average number of iterations of model M1 is significantly lower than M4, and it is concluded that the improvement of model M1 can effectively improve the convergence speed of neural network. The average accuracy rate of model M1 is significantly higher than other models, indicating that the improvement of model M1 can effectively improve the prediction accuracy of the model. The average f-measure

of model M1 is significantly is higher than other models, and it is concluded that the performance of model M1 is better than other models.

The performance comparison diagram of the M1, M5, M6 and M7 is shown in Fig. 2. The following conclusions can be drawn from the evaluation of precision and f-measure index. The performance of model M1 is better than that of model M7, which shows that the method of determining the input neuron of this model can effectively improve the performance of BP neural network. The performance of model M1 is better than that of model M6, which shows that the combination prediction can effectively determine the number of neurons in the hidden layer. The performance of model M1 is better than that of model M5, which shows that the improved genetic algorithm is one of the effective ways to improve the performance of BP neural network. In conclusion, M1 model can effectively solve the problem of customer churn prediction and analysis based on missing data.

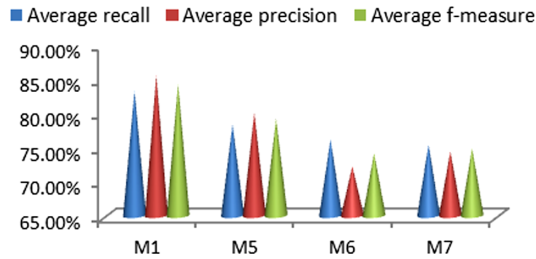


Fig. 2. Performance comparison chart of five models

4 Summary

This paper combines traditional statistical prediction methods and artificial intelligence prediction methods to propose a customer churn prediction analysis model based on missing data. In this model, the factor analysis method and data mining technology are used to generate key factor sets and their values related to predictive analysis problems based on missing data to form input neurons and their initial values. The number of hidden layer neurons was determined by combinatorial prediction. Using the improved genetic algorithm, the initial weight and threshold of BP network are determined. Finally, the prediction results and key attribute data related to the prediction results are generated for decision makers to analyze the problem. The experiment evaluates the model from the aspects of accuracy, precision, recall, and F1, which proves that the model is effective.

Acknowledgment. This work was financially supported by the national natural science foundation of China (61561055).

References

1. Barros, R.C., Basgalupp, M.P., Freitas, A.A., et al.: Evolutionary design of decision-tree algorithms tailored to microarray gene expression data sets. *IEEE Trans. Evol. Comput.* **18**(1), 873–892 (2014)
2. Dong, Y., Guo, H., Zhi, W., Fan, M., et al.: Class imbalance oriented logistic regression. In: *Cyber-Enabled Distributed Computing and Knowledge Discovery*, Shanghai, pp. 187–192 (2014)
3. Tang, L., Liu, H.: Bias analysis in text classification for highly skewed data. In: *Fifth IEEE International Conference on Data Mining*, New York, p. 4 (2005)
4. Zhang, Z., Cheng, H.: Clustering aggregation based on genetic algorithm for documents clustering. In: *IEEE World Congress on Computational Intelligence*, Hong Kong, pp. 3156–3161 (2008)
5. Immon, W.H.: *Building the data warehouse*. 北京: 机械工业出版社(2003)
6. Louis, A.C.: Data mining and causal modeling of customer behaviors. *Telecommun. Syst.* **21**(2–4), 349–358 (2002). <https://doi.org/10.1023/A:1020911018130>
7. Chum modeling for mobile telecommunications. <http://docs.salford-systems.com/--chumwinF08.pdf>
8. Ning, Y., Zheng, X.: Notice of retraction forecasting the natural forest stand age based on artificial neural network model. In: *Computer and Communication Technologies in Agriculture Engineering*, Chengdu, pp. 536–539 (2010)
9. Yang, Z., Shi, X.: An agent-based immune evolutionary learning algorithm and its application. In: *Intelligent Control and Automation*, Shenyang, pp. 5008–5013 (2014)
10. Chaudhuri, A., De, K.: A study of the traveling salesman problem using fuzzy self organizing map. In: *International Conference on Industrial and Information Systems*, Kharagpur, pp. 1–5 (2008)
11. Zhang, Y.: Cluster-based majority under-sampling approaches for class imbalance learning. In: *IEEE International Conference on Information and Financial Engineering*, Chongqing, 400–404 (2010)
12. Beigy, H., MeybodiM, R.: A learning automata-based algorithm for determination of the number of hidden units for three-layer neural networks. *Int. J. Syst. Sci.* **40**(1), 101–118 (2009)
13. Liu, B., Guo, HX.: *Matlab Neural Network Super Learning Manual*. People's Posts Telecommun. Publishing House, BeiJing (2014)
14. Ahmed, A.A., Maheswari, D.: Churn prediction on huge telecom data using hybrid firefly based classification. *Egypt. Inform. J.* **18**(3), 215–220 (2017)
15. Rizopoulos, D., Kuhn, M., Johnson, K.: Applied predictive modeling. *Biometrics* **74**(1), 383–393 (2018)



How to Evaluate Various Commonly Used Program Classification Methods?

Xinxin Qi, Yuan Yuan, Juan Chen^(✉), and Yong Dong

National University of Defense Technology, Changsha, China
{qixinxin19,yuanyuan,juanchen,yongdong}@nudt.edu.cn

Abstract. Understanding the characteristics of scientific computing programs has been of great importance due to its close relationship with the design and implementation of program optimization methods. Generally, scientific computing programs can be divided into three categories according to their computing, memory access and communication characteristics, namely compute-intensive, memory-intensive and communication-intensive, respectively. There are more than one commonly used program classification methods, particularly for compute-intensive and memory-intensive programs. In most cases, all kinds of classification methods have consistent results but occasionally different classification results also occur. Why are there occasionally inconsistent classification results and where? How to understand such inconsistencies and what is the reason behind that? We answer these questions by analyzing four representative program classification methods (IPC, MPKI, MEM/Uop and Roofline) on two platforms. Firstly, we discover some occasional inconsistency cases, the inconsistency from various indicators, the inconsistency from multi-phase characteristics and the inconsistency from various platforms, followed by some possible reasons. Secondly, we explore the impact of threshold settings on classification inconsistencies. All the experiment and analysis results and the data collected from other references prove that different classification methods have the same classification results in most cases but occasionally bring about inconsistencies especially for in-between programs that are between memory-intensive and compute-intensive programs, which have a bad impact on some optimization algorithms.

Keywords: Workload characterization · Program classification · Compute-intensive · Memory-intensive · In-between programs

1 Introduction

Understanding the characteristics of scientific computing programs has been important for the design and implementation of optimizations [14, 23, 24]. Due to

This work is supported in part by the Advanced Research Project of China under grant number 31511010203 and the Research Program of NUDT grant number ZK18-03-10.

various reasons, we have more than one program classification method to determine which type a program belongs to compute-intensive, memory-intensive or communication-intensive. Each classification method has its preference to some extent. *How to evaluate various classification methods* is what we are concerned about in this paper. We believe classification methods have consistent results in the majority of cases. But we find there are some inconsistencies in some cases. How to understand such inconsistent classification results and what is the reason behind that? When do the inconsistent classification results happen and are there any adverse influences? Because there are very few methods for identifying communication-intensive programs from the non communication-intensive programs [6, 15], in this article we only focus on the classification methods for compute-intensive and memory-intensive programs.

To better evaluate various program classification methods, we focus on the following two issues.

- *Issue 1.* There are about five classic classification methods and their preferences are a little bit different. *How to fairly compare them and discover the inconsistency between various classification methods?* Take **STREAM** as an example, [3] and [20] use three different classification methods to categorize **STREAM** and have the consistent results. Why these research works don't use the same classification method? On the one hand, they choose the most relevant classification method to assist the design of algorithms and models and to provide convenience for their research. For example, Reference [9] uses cache hit rates to classify programs because the cache hit rate is used to model memory bandwidth in their work. And Reference [3] uses Last Level cache miss rates as a classification indicator and builds a performance model with it to predict the ideal number of cores for OpenMP memory-intensive applications. On the other hand, some studies, such as the Roofline model [21], are devoted to providing easy-to-understand models for classification that offer performance guidelines. All these reasons lead to more than one classification method and different preferences. Another case is for **bzip2**, there is also more than one classification method. **bzip2** is categorized as memory-intensive in [12] but compute-intensive in [25]. Different classification methods produce inconsistent classification results, which reminds us to compare and identify the possible differences between these classification methods and explore the reasons for the inconsistency of the classification results.
- *Issue 2.* The threshold setting of classification indicators is important in almost all the classification methods. Even if the same classification indicator is used in different studies, the threshold setting is quite different. *What factors affect the threshold setting?* Classification methods sometimes depend on the hardware platform too much, which makes a lot of time spent on profiling and searching an appropriate threshold when changing to a different platform. Moreover, a multi-phase program usually has a distinct fluctuation in characteristics across different phases, which further makes the classification threshold unreasonable from the perspective of a whole program.

Furthermore, whatever threshold you set, how to cope with the programs whose indicator values are very close to the threshold? Those in-between programs are more likely to belong to neither compute-intensive nor memory-intensive. All these questions inspire us to explore the thresholds of classification indicators.

The corresponding solutions are explained as follows.

- *Solution to Issue 1.* We choose only the four representative classification methods, i.e. IPC (instructions per cycle), MPKI (cache misses per kilo instructions), MEM/Uop and Roofline, from existing numerous classification methods. Each of them has its own preference. To fairly compare these representative classification methods, we compare the differences between these classification methods by using a group of benchmarks and two platforms. We find out some occasionally inconsistent classification results and then figure out the possible reasons from the three aspects, classification indicators, multi-phase characteristics and hardware platforms (Sect. 3).
- *Solution to Issue 2.* We use the Roofline model to automatically identify the classification threshold so as to better show the impact of the hardware platform on the threshold setting. To evaluate the threshold setting, we take a multi-phase program as an example to analyze the influence of threshold settings on the classification results. In addition, we show more cases to prove that the threshold setting indeed has an impact on the in-between programs, which are between compute-intensive and memory-intensive (Sect. 4).

2 Experiment Platform

Hardware. Platform A contains 24 computing nodes, each of which consists of two Intel(R) Xeon(R) E5-2640 v2 CPUs. Platform B is a high performance computing cluster with 64 computing nodes. Each node is a dual-socket server, which consists of two Intel(R) Haswell(R) 10-core E5-2660 v3 CPUs. Table 1 provides a list of the hardware specifications in detail of platforms A and B.

Table 1. Characteristics of two hardware platforms

	Platform A	Platform B
CPU	Xeon E5-2640 v2	Haswell E5-2660 v3
Total threads	16	20
Total cores	16	20
Total sockets	2	2
GHz	2.0	2.6
Peak GFlops	256	416
Peak bandwidth	51.2 GB/s	68.4 GB/s
LLC	20 MB	25 MB
DRAM types	DDR3 800/1066/1333/1600 MHz	DDR4 1600/1866/2133 MHz

Benchmarks. We use two benchmark suites to evaluate different program classification methods, namely HPCC [18], NAS Parallel Benchmark (NPB) [4].

Tools and Metrics. We use Intel VTune [1] and Perf [2] to collect the status information of the relevant performance monitoring counters (PMCs) during the execution of the program. The performance monitoring events used in the paper are shown in Table 2.

Table 2. Performance monitoring events

Name	Performance monitoring event
IPC	<i>IPC</i>
Clock	<i>CPU_CLK_UNHALTED.THREAD</i>
Instructions	<i>INST_RETIRED.ANY</i>
Cache miss	<i>LLCMisses</i>
L3 cache misses	<i>MEM_LOAD_UOPS_RETIRED.L3_MISS</i>
Uops retired	<i>Uops_Retired.all</i>

3 Solution to Issue 1: Discover the Occasional Inconsistency

3.1 Overviews of Representative Program Classification Methods

It is the complexity in program classification that leads to inconsistencies in classification results. The complexity of program classification is not only caused by the characteristic behavior of the application itself, but also by the hardware platform running the application. In this paper, five commonly used program classification methods are selected from various existing program classification methods, i.e. IPC [11,14], MPKI [9–11] MEM/Uop [13], Roofline Model [21], and CCR (Communication-to-Computation Ratio) [6], and the basic ideas of these methods are explored. The rest of this section describes these classification methods in detail.

IPC. IPC metric uses the *instructions per clock cycle*, which reflects the number of instructions completed in each cycle. It is expected that the *IPC* for memory-intensive workloads is low due to the nature of long memory load latency while those that have small memory footprints tend to have very high *IPC* since all of their memory accesses are from the L2 or L3 caches [19]. In general, when *IPC* is greater than 1.0, it means the program is compute-intensive, and the program is memory-intensive conversely. The IPC metric is also used in the study of phase characteristics of programs [11,14]. Phase behavior in application characteristics has long been observed and exploited [7], where the program is divided into several phases and performance monitoring counters (PMCs) to monitor the

dynamic changes of each phase are used. Finally, the average *IPC* of each phase is calculated to determine the category of the program.

MPKI. MPKI metric uses *MPKI* as the indicator that indicates the LLC misses per kilo instructions to determine the memory boundness of programs. In [9] and [11], if *MPKI* of a program is greater than 5, it is a memory-intensive program. And if *MPKI* is greater than 10, it is highly memory-intensive.

MEM/Uop. The metric uses two PMCs, one of which is used to monitor the number of the retired micro-ops (Uops) to trigger the performance monitoring interrupt at the specified instruction granularity, and another is used to track memory bus transactions [13, 23]. The ratio *MEM/Uop* of memory bus transactions to the retired Uops is defined as a measure of memory boundness for each phase. The metric mentioned in [13] classifies according to the range of *MEM/Uop*, and defines six categories. Conceptually, category 1 corresponds to a highly compute-intensive mode and category 6 corresponds to a highly memory-intensive mode.

Roofline Model. The Roofline Model [21] is a 2-D graph model, which ties floating-point computing performance, operational intensity, and memory performance. The horizontal line in the Roofline model represents the peak floating-point performance, and the slash represents the peak memory bandwidth that the memory system of the computer can support under the given operation intensity. These two lines intersect at the point of peak computational performance and peak memory bandwidth. If we think of operational intensity as a column that hits the roof, either it hits the flat part of the roof, which means performance is compute-intensive, or it hits the slanted part of the roof, which means performance is ultimately memory-intensive. Based on the Roofline model, the Quadrant-Split model [16] is proposed for heterogeneous platforms. Different from the Roofline model, the Quadrant-Split model uses DRAM bandwidth as the x-axis instead of operational intensity.

CCR. The communication-to-computation ratio refers to the ratio of time to perform communication and computation [6].

3.2 Difference of Various Classification Methods

The differences of various program classification methods are explored from three aspects: classification types, applicable platforms and granularity.

- *Types.* Some program classification methods can be used to categorize programs as memory-intensive or compute-intensive and others are used to categorize programs as communication-intensive.
- *Platforms.* Some classification methods apply to CPU platforms, some apply to GPU platforms and others apply to the above two types of platforms.
- *Granularity.* There are two kinds of granularity involved in this paper, one is the whole program, the other is a part of the program, which we call phase. As mentioned in [13], we divide the program into many phases according to every 1000 instructions.

Table 3. Comparison of representative program classification methods

Method	Indicator	Type	Platform	Granularity
IPC	IPC	Memory, compute	CPU, GPU	Phase
MPKI	LLC misses	Memory, compute	CPU, GPU	Phase
MEM/Uop	<i>UOPS_RETIRED</i> , <i>BUS_TRAN_MEM</i>	Memory, compute	CPU	Phase
Roofline	<i>GFLOPS</i> , DRAM bandwidth	Memory, compute	CPU	Program
CCR	Computation-to- communication ratio	Communication	CPU, GPU	Program

Table 3 theoretically compares the differences between several classification methods. Classification indicators, types, applicable platforms and classification granularity of selected classification methods are listed in Table 3. According to Table 3, the five classification methods use completely different indicators, which have been introduced in Sect. 3.1. In terms of classification types, the first four metrics in Table 3 are used to determine the compute intensiveness and memory intensiveness of the program, while the CCR is used to determine the communication intensiveness. From the perspective of applicable platforms, the three methods, IPC, MPKI and CCR metrics, apply to both CPU and GPU platforms, while Mem/Uop and Roofline models do not apply to GPU platforms, because the GPU platform does not support the measurement of relevant indicators. We can use IPC, MPKI, and MEM/Uop metrics to assign a category to each phase (which can be extended to the whole program). However, the Roofline model and CCR are only applicable to the classification of the whole program, and these methods cannot describe the dynamic behavior of the program during execution.

3.3 Exploration of Occasional Classification Inconsistency

We implement several program classification methods on the hardware platform mentioned in Sect. 2. To compare the relationship between different classification methods, here we only compare the classification methods designed for memory-intensive and compute-intensive programs, instead of communication-intensive programs, because there are no other representative classification methods for communication-intensive programs other than CCR. We use Perf and Intel(R) VTune to measure PMCs when the program executes. To implement the phase methods, we set the sampling interval as 1000 instructions. We explore and compare experiment results from the three aspects: indicators, multi-phase characteristics and hardware platforms.

Inconsistency 1: Indicators. Table 4 lists the measured values of classification indicators of HPCC and NPB benchmarks with the same size of and the

Table 4. Classification results for NPB and HPCCL on platform A. [C] represents compute-intensive and [M] represents memory-intensive.

Benchmark	IPC	MPKI	MEM/Uop
Thresholds	1.0 [M→C]	10.0 [C→M]	0.0005 [C→M]
RandomAccess	0.27 [M]	147.99 [M]	0.0730 [M]
STREAM	0.50 [M]	43.85 [M]	0.0066 [M]
CG	0.99 [M]	9.56 [C]	0.0030 [M]
EP	1.09 [C]	0.55 [C]	0.0000 [C]
FFT	1.52 [C]	8.94 [C]	0.00053 [M]
SP	1.59 [C]	10.26 [M]	0.0006 [M]
PTRANS	1.65 [C]	4.25 [C]	0.0017 [M]
LU	1.90 [C]	3.42 [C]	0.0000 [C]
BT	2.12 [C]	10.78 [M]	0.0002 [C]
MG	2.41 [C]	3.50 [C]	0.0002 [C]
FT	2.49 [C]	4.64 [C]	0.0002 [C]
DGEMM	2.55 [C]	6.35 [C]	0.0001 [C]

same number of processors on the platform A. The letters in square brackets are the categories of programs, where C represents a compute-intensive type, and M represents a memory-intensive type. We set the thresholds of each metric according to the threshold setting methods mentioned in the classification methods described in Sect. 3.1. The thresholds of IPC, MPKI, and MEM/Uop metrics are as follows. $IPC_{threshold} = 1.0$, $MPKI_{threshold} = 10.0$, and $MEM/Uop_{threshold} = 0.0005$. When IPC of a program higher than $IPC_{threshold}$, it is compute-intensive and memory-intensive conversely. When $MPKI$ of a program higher than $MPKI_{threshold}$, it is memory-intensive and compute-intensive conversely. When MEM/Uop of a program higher than $MEM/Uop_{threshold}$, it is memory-intensive and compute-intensive conversely. From Table 4, we find that the results of different classification methods are occasionally inconsistent. That shows the selection of classification indicators has an impact on the classification results. Can we consider most of the classification results of a program as the right classification results and other inconsistent classification results as the wrong one?

To better observe the relationships between the classification indicators, we rank programs according to the order of IPC from low to high as Table 4 shows. Generally, the program is considered to be compute-intensive when IPC is high and $MPKI$ is low, and to be memory-intensive when IPC is low and $MPKI$ is high. So, if we sort the programs by the order of IPC from low to high, we expect to have the values of $MPKI$ to be ordered from high to low. However, there are some exceptional values, such as SP ($IPC \uparrow$), CG ($MPKI \downarrow$), BT ($MPKI \uparrow$), FFT ($MEM/Uop \uparrow$), PTRANS ($MEM/Uop \uparrow$).

Observation 1: Sometimes only IPC fails to have an accurate classification result. Combining IPC with MPKI is expected to have a more accurate classification results. According to the MPKI and MEM/Uop metrics, SP is memory-intensive. But IPC metric has the opposite result. That is an example to show only IPC sometimes cannot have the accurate program categories. References [11] and [5] also mentioned this, in which both IPC and MPKI metrics are used to determine the category of a program. When we comprehensively consider IPC and MPKI metrics, we think the following principles should be followed. i) A program with low IPC and high MPKI is memory-intensive, while a program with high IPC and low MPKI is compute-intensive. ii) When a program has low IPC and low MPKI, it indicates that the program itself contains very little memory access and very few instructions for computation, and we consider the program to be compute-intensive. iii) If a program has the characteristics of high MPKI and high IPC, it means that the program contains a lot of instructions for computation and has intensive memory access. We don't exclude the fact that programs are both compute-intensive and memory-intensive, which has been mentioned in [25]. Therefore, sometimes the program will be subdivided into many phases, and the categories of each phase will be judged.

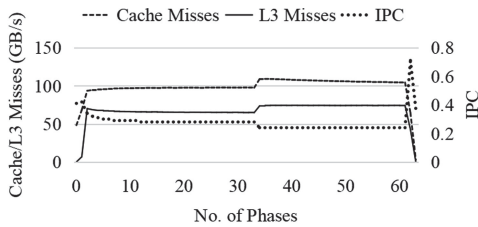
Observation 2: Those programs whose metrics are very close to thresholds easily have wrong classification results. The MPKI values of CG, SP and BT are very close to MPKI thresholds. The MEM/Uop values of FFT (0.00053) is very close to its threshold (0.0005). Classification error possibilities for these programs are higher than other programs. A small fluctuation when setting thresholds will change classification results. The metric (MPKI or MEM/Uop) value near to the threshold exactly proves the program has an ambiguous type feature that falls in between compute-intensive and memory-intensive. Aided by IPC metric, CG, BT and FFT can be classified to the right program types.

Observation 3: The use of MEM/Uop metric results in the inclusion of "concurrent execution" information for the program, which further improves the accuracy of the program classification results. In our experiments, the classification results of both IPC and MPKI show that PTRANS is a compute-intensive program. But Reference [8] mentioned that PTRANS is a memory-intensive program that involves a lot of access to remote memory. Neither IPC nor MPKI can reflect the concurrent execution information of the program, which leads to inaccurate classification. Compared with IPC and MPKI, MEM/Uop can provide additional concurrent execution information, because MEM/Uop is the ratio of memory bus transactions to uops retired, which implies the ratio of uops to instructions reflects the concurrent execution information of the program. The more detailed and accurate information provided by MEM/Uop can help the program classification more accurately.

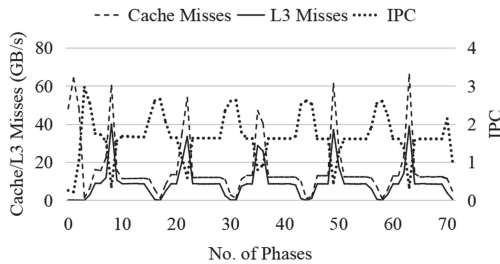
Inconsistency 2: Multi-phase Characteristics. For some complex programs, their program behaviors can be easily divided into different phases (called multi-phase programs) instead of only one phase (called single-phase programs).

Figure 1 shows an example of these two kinds of programs. `RandomAccess` in Fig. 1(a) is a single-phase program where its performance indicators stable during the whole execution process. `PTRANS` in Fig. 1(b) is a multi-phase program where its performance characteristics are separated by several different phases.

Due to the complexity of multi-phase programs, we cannot classify them by a unique type. Some phases of multi-phase programs perhaps belong to the compute-intensive type and other phases maybe can be classified as memory-intensive. There exists some phased classification methods, such as `MPKI` [13] and phased `IPC` [14]. `MPKI` has the feature of phased classification methods because `MPKI` counts the LLC misses per 1000 instructions. When we adopt these phased classification methods instead of classification methods for the whole program, will phased classification methods have the same results? We choose $MPKI_{phase}$ and $MPKI_{program}$ as examples to illustrate the impact of multi-phase characteristics on the classification results. $MPKI_{phase}$ refers to a classification method for phases, which are divided according to every 1000 instructions. $MPKI_{program}$ refers to a classification method for the whole program. Table 5 shows a comparison for $MPKI_{phase}$ and $MPKI_{program}$. Nearly all the programs have the same classification results for $MPKI_{phase}$ and $MPKI_{program}$ except `FFT`.



(a) `RandomAccess` with a single phase



(b) `PTRANS` with multiple phases

Fig. 1. Phase behavior comparison for a single-phase program (`RandomAccess`) and a multi-phase program (`PTRANS`).

Table 5. Classification results of phased MPKI and traditional MPKI. The value of 100/8 represents 100 phases for compute-intensive and 8 phases for memory-intensive.

Benchmark	Number of phases (C/M)	$MPKI_{phase}$	$MPKI_{program}$
DGEMM	100/8	C	C
RandomAccess	1/63	M	M
STREAM	1/29	M	M
FFT	10/20	M	C
PTRANS	63/8	C	C

Every program in Table 5 is divided into many phases, such as DGEMM with 109 phases, RandomAccess with 64 phases, STREAM with 30 phases, FFT with 30 phases and PTRANS with 71 phases. The second column in Table 5 shows the number of phases. Furthermore, these phases are separated into two types, one for compute-intensive and the other for memory-intensive. For example, DGEMM is identified by 100 phases for compute-intensive and 8 phases for memory-intensive. $MPKI_{phase}$ determines the whole program type by the numbers of compute-intensive phases and memory-intensive phases. If the number of compute-intensive phases is greater than the number of memory-intensive phases, the program is regarded as compute-intensive, otherwise it is a memory-intensive program.

FFT is classified as the memory-intensive when using $MPKI_{phase}$ but the compute-intensive when using $MPKI_{program}$. As can be seen from Table 5, FFT has 1/3 possibility for the compute-intensive and has 2/3 possibility for the memory-intensive. The phased classification method can find out the potential computing density and memory density of FFT. However, classification methods for the whole program can not fully reveal all the possible computation and memory access behaviors of a program, which leads to inaccurate or even wrong classification results.

Inconsistency 3: Platform. Different hardware platforms have different physical memory bandwidth and physical floating-point computing performance, so the characteristics of the same program running on different hardware platforms may be different. The different thresholds for various platforms are expected. We compare the same classification methods on two platforms (Platform A and Platform B) to explore the impact of the hardware platform on program classification as Table 6 shows. To more intuitively compare the two platforms, we rank programs in Table 6 according to the order of IPC of platform B from low to high. We set the thresholds of IPC , $MPKI$, and MEM/Uop on platform B as follows. $IPC_{threshold} = 1.0$, $MPKI_{threshold} = 10.0$, and $MEM/Uop_{threshold} = 0.03$, respectively.

The inconsistencies caused by hardware platform are bolded in Table 6, BT and CG. See more explanations as follows.

Table 6. Classification results for NPB and HPCC on two platforms.

Benchmark	IPC		MPKI		MEM/Uop	
	Platform A	Platform B	Platform A	Platform B	Platform A	Platform B
Thresholds	1.0	1.0	10.0	10.0	0.0005	0.03
RandomAccess	0.27 [M]	0.62 [M]	147.99 [M]	131.20 [M]	0.073 [M]	0.084 [M]
STREAM	0.5 [M]	0.74 [M]	43.85 [M]	17.55 [M]	0.0066 [M]	0.035 [M]
EP	1.09 [C]	1.21 [C]	0.55 [C]	0.31 [C]	0.0000 [C]	0.001 [C]
MG	2.41 [C]	1.65 [C]	3.5 [C]	3.22 [C]	0.0002 [C]	0.0009 [C]
FFT	1.52 [C]	1.69 [C]	8.94 [C]	9.84 [C]	0.0005 [M]	0.01 [M]
PTRANS	1.65 [C]	1.81 [C]	4.25 [C]	3.68 [C]	0.0017 [M]	0.012 [M]
DGEMM	2.55 [C]	1.86 [C]	6.35 [C]	7.11 [C]	0.0001 [C]	0.0063 [C]
FT	2.49 [C]	2.11 [C]	4.64 [C]	4.73 [C]	0.0002 [C]	0.0027 [C]
SP	1.59 [C]	2.15 [C]	10.26 [M]	1.03 [C]	0.0006 [M]	0.003 [M]
LU	1.9 [C]	2.22 [C]	3.42 [C]	0.93 [C]	0.0000 [C]	0.0035 [C]
BT	2.12 [C]	2.29 [C]	10.78 [M]	7.1 [C]	0.0002 [C]	0.0022 [C]
CG	0.99 [M]	2.38 [C]	9.56 [C]	2.1 [C]	0.0030 [M]	0.044 [M]

- The first inconsistency happens on program BT when using the MPKI metric. BT is classified as compute-intensive on platform B because of lower MPKI than $MPKI_{threshold}$. According to the *Observation 2* in Subsection *inconsistency 1:indicators*, the MPKI value of BT is close to the threshold, which makes BT have a higher risk of having a wrong classification result. BT on platform A has a 50% probability of belonging to the compute-intensive and the memory-intensive. From this perspective, the classification results on both platforms A and B are consistent.
- The second inconsistency is about CG. The number of compute-intensive type is the same as that of memory-intensive type. It is hard to identify which type CG belongs to. CG shows the half-half intensive characteristics. The following two facts prove this. Firstly, as mentioned in Subsection *inconsistency 1:indicators*, the IPC of CG (0.99) is very close to the threshold (1.0), which means CG has a 50% probability of belonging to compute-intensive and memory-intensive. Furthermore, platform B has higher peak floating-point computing performance and faster CPUs, which contributes higher IPC, so CG becomes compute-intensive by IPC metric when the platform changes from A to B. Secondly, both MPKI and MEM/Uop metrics remain the same classification results for two platforms. One explanation is that CG not only has a high instruction number per cycle and but also has a high LLC misses per kilo instructions. To summarize, we prefer to classify CG as an in-between type (half-half intensive) between the compute-intensive and the memory-intensive. There indeed exists such a category, neither typical compute-intensive nor typical memory-intensive [22]. The authors in [22] take the in-between programs as a special case to optimize and have a quite different result compared to the traditional classifications.

The above analysis shows it is important and difficult to find out the dividing line (thresholds) between the compute-intensive and the memory-intensive. In the next section, we will discuss the problem of exploring the threshold.

4 Solution to Issue 2: Explore the Threshold

What factors influence the threshold setting? (Sect. 4.1) And if there are no clear dividing lines for some half-half intensive programs, how to consider these in-between programs? (Sect. 4.2).

4.1 The Impact of Thresholds

We use the Roofline model to analyze the impact of various platforms on classification thresholds. The Roofline model is built in terms of hardware specifications, which is sensitive to the platform and also threshold changes. Figure 2 uses NPB benchmarks to show Roofline model classification results and threshold changes. Roofline model clearly marks the peak floating-point computing performance with a horizontal line (256 GFLOPS, 416 GFLOPS) and the peak memory bandwidth with an oblique line (51.2 GB/s, 68.4 GB/s) on two platforms. Roofline model also can describe the execution feature of a program, such as operational intensity (X-axis). A group of lines parallel to the Y-axis represents different NPB programs with various operational intensities. The joint point of the horizontal line and the oblique line divides programs into two types, the compute-intensive (black dotted lines to the right of red line) and the memory-intensive (black dotted lines to the left of red line). The two platforms have the same classification results. However, The place of joint points is different for the two platforms. The different platforms result in various classification thresholds of operational intensity, 5.0 flops/byte on platform A and 6.04 flops/byte on platform B.

Different from the Roofline model, IPC, MPKI, and MEM/Uop methods cannot automatically identify appropriate thresholds to categorize programs, which aggravates the difficulty of program classification. Some other research works also show the threshold will change when the platform changes. For example, IPC threshold in [11] is 1.0, while the IPC threshold becomes 1.2 in [14] when the platform changes. Usually, researchers decide the threshold by their experiences and sometimes have uncertain optimization results. In Sect. 3, we show that the selection of thresholds has an impact on the classification results for the two types of programs, single-phase and multi-phase programs.

For a multi-phase program, whether a classification threshold is appropriate will greatly affect the classification result of each phase and further affect the category of the whole program. Figure 3 shows an example. The blue dashed horizontal line and the red one shows the two thresholds, $MPKI_{threshold} = 10$ and $MPKI_{threshold} = 50$. When the threshold of $MPKI$ is increased from 10 to 50, the number of memory-intensive (compute-intensive) phases will be changed from 29 (1) to 15 (15), and the category will be changed from the memory-intensive to the non-memory-intensive, an in-between type. The categories of

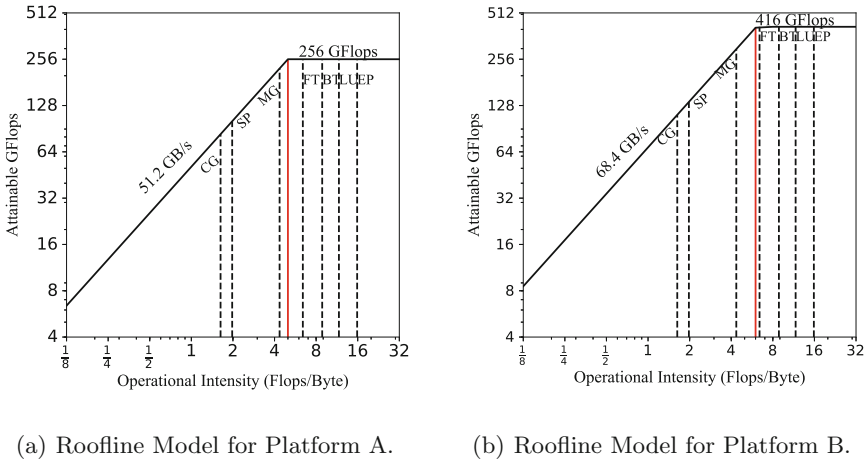


Fig. 2. Comparison of roofline model of platforms A and B (Color figure online)

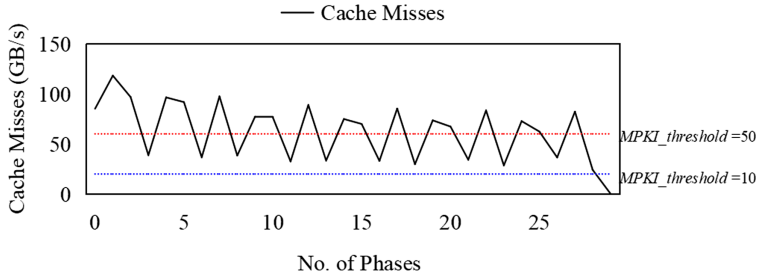


Fig. 3. The impact of threshold on multi-phase program *STREAM*.

in-between programs is really hard to be identified. The next section will reveal more cases for in-between programs.

4.2 Discussions About In-between Programs

One case for an in-between program is that it involves not only a lot of computation, but also a lot of memory access. The programs can be regarded as both the compute-intensive and the memory-intensive from this point of view. For such programs, classification results are often inconsistent. For example, *bzip2* is categorized as a compute-intensive program in [17,25], but as an in-between program in [12].

Another case occurs when the metric value is very close to the threshold. From Table 4, *CG* is categorized as a compute-intensive program because its lower $MPKI$ than $MPKI_{threshold}$. But $MPKI$ of *CG* is very close to the $MPKI_{threshold}$, which shows *CG* also involves intensive access to memory.

Classifying the in-between programs mainly depends on what optimization goal you focus on. There is no unique rule. Exploiting more program behavior characteristics and combining them with the metrics you are concerned about maybe will be helpful to the optimization methods. For example, Reference [22] used four in-between programs, namely `canneal`, `facesim`, `ferret`, and `streamcluster`, which they refer to as weakly memory-intensive programs in the paper. Initially, they categorized these four in-between programs as memory-intensive and adopted optimizations that are designed for memory-intensive programs to them. However, they found that the optimization methods brought an average performance loss of 2.9% to these four in-between programs, and the maximum performance loss could reach to 5.6% (`facesim`). Although these four benchmarks were classified as memory-intensive programs, the memory bandwidth was relatively low. So they turned to adopt the optimizations for compute-intensive programs. It is surprising that the average performance improvement for these four in-between programs was 9.9%, and the maximal performance improvement was up to 13.2% (`canneal`). Thus it can be seen that the classification of in-between programs is extremely difficult, and a deeper understanding of the characteristic behavior of programs can lead to flexible and effective optimization of these programs.

5 Conclusions

Understanding the characteristics of scientific computing programs has always been important for the design and implementation of optimizations. In this paper, we explore and compare several representative classification methods from existing classification methods theoretically and experimentally. In most cases, all kinds of classification methods have consistent results but occasionally different classification results also occur. To figure out why there are inconsistent classification results and what are the reasons behind that? We analyze the causes of inconsistencies from three aspects: indicators, multi-phase characteristics and platforms, and we gain some valuable observations. To further understand these inconsistencies, we focus on the selection of thresholds and analyze its influencing factors. It is found that in-between programs are easily affected by the threshold setting and produce the wrong classification, which has a bad impact on optimizations.

References


1. Intel® vtune™ amplifier (2019). <https://software.intel.com/en-us/vtune>
2. Perf (2019). https://perf.wiki.kernel.org/index.php/Main_Page
3. Alcaraz, J., Sikora, A., Cesar, E.: Dynamic tuning of openmp memory bound applications in multisocket systems using mate. In: Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP 2018. Association for Computing Machinery, New York (2018). <https://doi.org/10.1145/3229710.3229748>. <https://doi-org-s.nudtproxy.yitlink.com/10.1145/3229710.3229748>

4. Bailey, D., et al.: The NAS parallel benchmarks. *Int. J. High Perform. Comput. Appl.* **5**, 63–73 (1991). <https://doi.org/10.1177/109434209100500306>
5. Begum, R., Werner, D., Hempstead, M., Prasad, G., Challen, G.: Energy-performance trade-offs on energy-constrained devices with multi-component DVFs. In: 2015 IEEE International Symposium on Workload Characterization (IISWC) (2015)
6. Crovella, M., Bianchini, R., LeBlanc, T., Markatos, E., Wisniewski, R.: Using communication-to-computation ratio in parallel program design and performance prediction. In: *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pp. 238–245 (1992)
7. Denning, P.J.: The working set model for program behavior. *Commun. ACM* **11**(5), 323–333 (1968)
8. Ge, R., Zou, P., Feng, X.: [IEEE 2017 46th International Conference on Parallel Processing (ICPP) - Bristol, United Kingdom (14 August 2017–17 August 2017)] 2017 46th International Conference on Parallel Processing (ICPP) - Application-Aware Power Coordination on Power Bounded Numa Multicore, pp. 591–600 (2017)
9. Hashemi, M., Mutlu, O., Patt, Y.: Continuous runahead: transparent hardware acceleration for memory intensive workloads, pp. 1–12 (2016). <https://doi.org/10.1109/MICRO.2016.7783764>
10. Hashemi, M., Mutlu, O., Patt, Y.N.: Continuous runahead: transparent hardware acceleration for memory intensive workloads (2016)
11. Hashemi, M., Patt, Y.N.: Filtered runahead execution with a runahead buffer. In: *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 358–369 (2015)
12. Huang, S., Feng, W.: Energy-efficient cluster computing via accurate workload characterization. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18–21 May 2009 (2009)
13. Isci, C., Contreras, G., Martonosi, M.: Live, runtime phase monitoring and prediction on real systems with application to dynamic power management, pp. 359–370 (2006). <https://doi.org/10.1109/MICRO.2006.30>
14. Jang, H., Lee, J., Kong, J., Suh, T., Chung, S.: Leveraging process variation for performance and energy: in the perspective of overclocking. *IEEE Trans. Comput.* **63**, 1 (2014). <https://doi.org/10.1109/TC.2012.286>
15. Chen, J., et al.: Analyzing time-dimension communication characterizations for representative scientific applications on supercomputer systems. *Front. Comput. Sci.* **13**(6), 1228–1242 (2019)
16. Konstantinidis, E., Cotronis, Y.: A practical performance model for compute and memory bound GPU kernels (2015). <https://doi.org/10.1109/PDP.2015.51>
17. Loew, J., Ponomarev, D.: Two-level reorder buffers: accelerating memory-bound applications on SMT architectures. In: 2008 37th International Conference on Parallel Processing, pp. 182–189 (2008)
18. Luszczek, P., et al.: The HPC challenge (HPCC) benchmark suite, p. 213 (2006). <https://doi.org/10.1145/1188455.1188677>
19. Maron, B., Chen, T., Vianney, D., Olszewski, B., Kunkel, S., Mericas, A.: Workload characterization for the design of future servers. In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium*, pp. 129–136 (2005). <https://doi.org/10.1109/IISWC.2005.1526009>

20. Tikir, M.M., Carrington, L., Strohmaier, E., Snively, A.: A genetic algorithms approach to modeling the performance of memory-bound computations. In: SC 2007: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–12 (2007)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**, 65–76 (2009). <https://doi.org/10.1145/1498765.1498785>
22. Wu, F., et al.: A holistic energy-efficient approach for a processor-memory system. *Tsinghua Sci. Technol.* **24**(4), 468–483 (2019)
23. Wu, Q., et al.: A dynamic compilation framework for controlling microprocessor energy and performance. In: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (2005)
24. Dong, Y., Chen, J., Tang, Y., Wu, J., Wang, H., Zhou, E.: Lazy scheduling based disk energy optimization method. *Tsinghua Sci. Technol.* **25**(2), 203–216 (2020)
25. Zhou, H., Conte, T.M.: Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Trans. Comput.* **54**(7), 897–912 (2005)



A Performance Evaluation Method for Machine Learning Cloud

Yue Zhu¹✉, Shazhou Yang², Yongheng Liu³, Longfei Zhao⁴,
and ZhiPeng Fu³

¹ Beijing Institute of Technology, Beijing 100081, China
gongchensu@gmail.com

² School of Computer Science, National University of Defense Technology,
Changsha 410073, China

³ Peng Cheng Laboratory, Shenzhen 518055, China

⁴ The Fifth Electronic Research Institute of MIIT, Guangzhou 510610, China

Abstract. In recent years, the application of machine learning algorithm is more and more extensive, and the combination of cloud platform and machine learning algorithm is closer. With the popularity of cloud platform, more and more cloud platform providers, the comparison of performance of different cloud platforms becomes crucial. The cloud platform performance benchmark can provide a relatively objective reference for consumers, However, the current mature cloud platform performance benchmarks cannot meet the requirements of testing the machine learning capabilities of cloud platforms, while the recent ones just only test the performance of machine learning. Based on the previous cloud platform performance testing methods, this paper designed a cloud platform performance evaluation method for machine learning applications based on the combination of AI-based testing benchmark and CPU-based testing benchmark, which can not only evaluate the performance of cloud platform in terms of CPU, but also test the performance of cloud platform in terms of GPU, running machine learning algorithms.

Keywords: Evaluation method · Cloud benchmark · Machine learning

1 Introduction

Cloud computing has taken off since Amazon launched Amazon webservices cloud computing in 2005 and launched cloud services such as EC2 the following year. With the close combination of big data and cloud computing in 2013, cloud computing is even stronger. It is an inevitable trend to process big data with artificial intelligence algorithms such as neural network. Cloud computing, big data and artificial intelligence have become the basis of modern business [1]. Internet companies such as Google, Amazon and China's Baidu are all working to combine cloud platforms with artificial intelligence. Earlier, DELOITTE

Supported by Peng Cheng Laboratory.

Global predicted that enterprises will accelerate the use of cloud-based AI software and services in 2019. By 2020, the penetration rate of integrated AI and cloud-based AI platforms among companies using AI software will reach 87% and 83%, respectively [2]. The combination of cloud platforms and artificial intelligence will be inevitable, while cloud platforms that support machine learning algorithms will become mainstream. Just like the development of cloud platforms, cloud platforms supporting machine learning applications need to have corresponding benchmark standards to evaluate the performance of AI cloud platforms, to promote the development of artificial intelligence cloud platform. Based on Spec Cloud, an evaluation method for cloud platform performance, and MLPerf benchmark for AI application performance testing, this paper designs a benchmarking method for cloud platform supporting machine learning application, which can make a complete evaluation of cloud platform performance including the performance in terms of AI application. And the test experiment on the domestic cloud platform proves that the performance evaluation method designed in this paper can comprehensively evaluate the performance of the cloud platform in terms of CPU and GPU.

2 Correlational Research

Performance benchmarks for infrastructure as a service cloud platform can be broadly divided into ML benchmark, which is still in development, and Compute benchmark, which is more mature. Google's own cloud platform benchmark, PerfKit Benchmarker [3], is a mature benchmark of Compute benchmark. But PerfKit contains a lot of redundant performance data that does not measure the performance of the cloud platform's machine learning algorithms. The most authoritative benchmarks for evaluating cloud platform performance are those published by SPEC [4] and TPC [5]. TPC can be regarded as an application-level benchmark on the cloud platform, primarily as a basis for evaluating the actual performance of standard transaction programs on machines published on IaaS. SPEC can evaluate the performance of the cloud platform, providing a fair and effective set of metrics for the cloud platform market so that business customers can choose the right cloud platform based on these results [6]. Although Spec Cloud is a relatively mature benchmark for evaluating the performance of Cloud platforms, it does not consider enough about the performance testing of AI applications and cannot reflect the support of Cloud platform for AI applications. ML benchmark which can be used to test cloud platform is MLPerf [7], the industry's first objective artificial intelligence benchmark suite, combines the standards of many previous benchmarks, such as the program by SPEC, the way SORT is used to compare and foster new ideas, DeepBench's assessment of software applications, and DAWNBench's precision standards. Although MLPerf can also help facilitate the development of machine learning algorithms, it cannot evaluate the performance of cloud platform in terms of CPU. The cloud platform performance evaluation method designed in this paper fully considers the comprehensiveness of performance evaluation. It can not only test the CPU

performance and scalability of the cloud platform, but also test its ability to support artificial intelligence algorithms.

3 Design of Cloud Platform Performance Evaluation Method

This article's cloud platform performance evaluation method of design can be divided into three aspects: the first is performance evaluation for the CPU hardware of cloud platform, the second is performance evaluation for GPU of the cloud platform, finally is the resource dispatching capability evaluation of the cloud platform itself. The purpose of this article is to design a method that can comprehensively evaluate the performance of cloud platform from these three aspects. One of the mature Cloud platform performance benchmarks is the SPEC Cloud benchmark. The SPEC Cloud is based on the CBTOOL tool [8] and can be deployed automatically to test the CPU performance of the Cloud by using Apache Cassandra's YCSB and Apache Hadoop structured k-means workloads, to test the performance of the cloud platform on the CPU side from read, write, and compute. At the same time, SPEC Cloud tests the elasticity and scalability of the Cloud platform by constantly adding random read and write loads to the Cloud platform during the scale-out phase. The SPEC Cloud emulates the standard social media NoSQL database application scenario, thus partially demonstrating its support for AI algorithmic performance testing in data storage. But SPEC Cloud still does not visually evaluate the capabilities of the Cloud platform in terms of machine learning algorithms. The new benchmark suite, MLPerf, can measure a range of machine learning workloads. It can test the performance of a cluster in machine learning, but it can only measure the performance of a machine learning algorithm, not the overall performance of a cloud platform. Therefore, this paper intends to design a cloud platform performance evaluation method that can automate to deploy loads to evaluate the cloud platform performance and get an evaluation result more comprehensively. Based on the main framework of SPEC Cloud, this paper designs an evaluation method based on SPEC Cloud that can measure the performance of AI Cloud platform. The main structure of this evaluation method is shown in the figure, which is divided into two parts: control node and workload. The control node will automate the entire benchmarking process, deploy the workload in the cloud platform, test in two phases, and finally collect and generate the final report by the control node. The workload to test the performance of the cloud platform is a virtual machine cluster deployed on the cloud platform to perform algorithm tasks. To automate the performance evaluation process, the control node needs to meet the following functions:

- Starts and stops the application instances and workload generator.
- Collects and aggregate the results.
- Determines if a run was successful.
- Generate a full disclosure report.

SPEC Cloud benchmarks use the Cloud Rapid Experimentation and Analysis tool (CBTOOL) to automate the benchmarking process, and this article continues to use this tool to provide the ability to schedule and launch benchmarks. It is an Apache 2.0-licensed cloud benchmarking tool that exposes an API that is used by the baseline and scale-out drivers to perform the two phases of the benchmark and then use the report generator to generate reports for the baseline and scale-out phases of the benchmark.

3.1 Workload

In the Cloud platform performance evaluation method designed in this paper, there are three workloads, among which YCSB and k-means workloads are the two workloads adopted by SPEC Cloud itself, and an MLPerf is added as the third workload. The three workloads are compute-intensive, I/O intensive and machine learning, testing the performance of the cloud platform from three aspects: CPU reading or writing and computing, and artificial intelligence algorithm running.

I/O Intensive Workload: Yahoo! Cloud Serving Benchmark (YCSB) with Apache Cassandra. Social network sites are one of the most popular applications for large cloud computing. Social network sites contain many types of computing services, of which NoSQL database is a key component and is I/O intensive. With Apache 2.0 support, YCSB can simulate many types of database transactions, including a read-based transaction that mixes most typical social media database activity. This article uses the YCSB workload D (95% read, 5% insert) used by SPEC Cloud IaaS 2018 benchmarks to simulate simple social network user activity. For NoSQL databases, the Apache Cassandra database was used as the underlying NoSQL database because the Apache Cassandra database proved to be more sensitive to I/O and CPU resource constraints during benchmark development. Figure 1 shows the structure of the YCSB application instances in the designed benchmark. The YCSB driver instance generates load on the Cassandra cluster. The Cassandra cluster comprises six instances. Together, these seven instances comprise the YCSB application instance for the benchmark. The choice of six instances for Cassandra represents a tradeoff between a trivial cluster size (e.g., two) and large cluster sizes (e.g., twenty), and having more than one workload generators to saturate the cluster, which will be required for large cluster sizes. Cassandra supports two types of nodes in its cluster configuration, namely, seeds and data nodes. Seeds during startup work to discover the other seeds/data nodes that make up the cluster [Reference: CassandraSeedsOne]. One design option was to use three seeds and three data nodes. The data nodes take a non-deterministic time to join the cluster. Moreover, multiple data nodes joining at the same time is potentially problematic. The Cassandra documentation at the time of benchmark development, recommended a gap of two minutes between multiple 9 Cassandra data nodes that join an existing cluster [Reference: CassandraAddDataNodes]. Therefore, YCSB application instance uses six seeds as the six Cassandra instances.

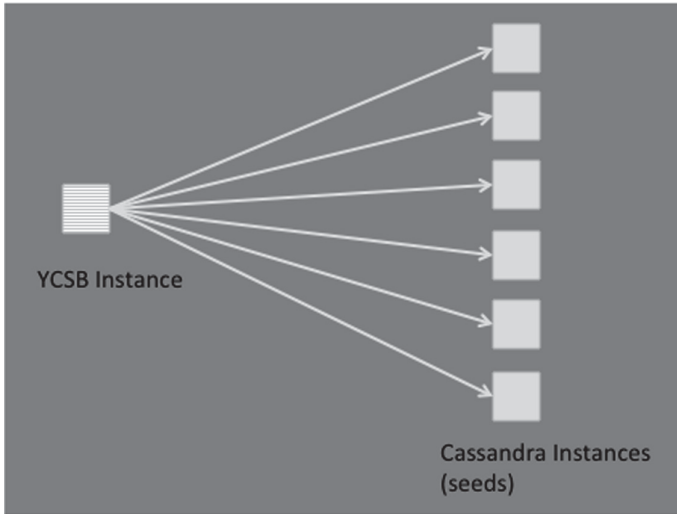


Fig. 1. YCSB/Cassandra application instance.

Compute-Intensive Workload - K-Means with Apache Hadoop. The K-Means algorithm is a popular clustering algorithm used in machine learning. SPEC Cloud IaaS 2018 Benchmark uses Intel HiBench K-Means implementation [Reference: HiBenchIntro]. K-Means is one of the nine Hadoop workloads that are part of the HiBench suite. HiBench was selected as the benchmark suite as it provides multiple Hadoop workloads and has a uniform interface for running these workloads. HiBench uses Apache Mahout [Reference: ApacheMahout] for K-Means implementation. The HiBench K-Means workload was selected based on its range of workload models, and built-in data generator to drive the load. The workload comprises a Hadoop name node instance, which also runs the Intel HiBench workload driver. The data is processed on five Hadoop data nodes. Together, these six instances comprise the K-Means application instance in SPEC Cloud IaaS 2018 Benchmark. Figure 2 shows the logical architecture of K-Means application instance in SPEC Cloud IaaS 2018 Benchmark.

Artificial Intelligence Training Workload. This is a newly developed method to evaluate the performance of hardware and software running artificial intelligence algorithm training and reasoning. The purpose of training or reasoning test is to measure how quickly a system can train the model to achieve target quality or complete inference, so the final unit of measurement is time. MLPerf has two types of benchmarks, closed and open. Closed versions require the same preprocessing, model, and training methods to be used as reference implementations, primarily to compare performance across platforms. MLPerf allows submitters to reimplement the reference implementations aiming to encourage innovation in software as well as hardware. MLPerf has two Divisions, one is the

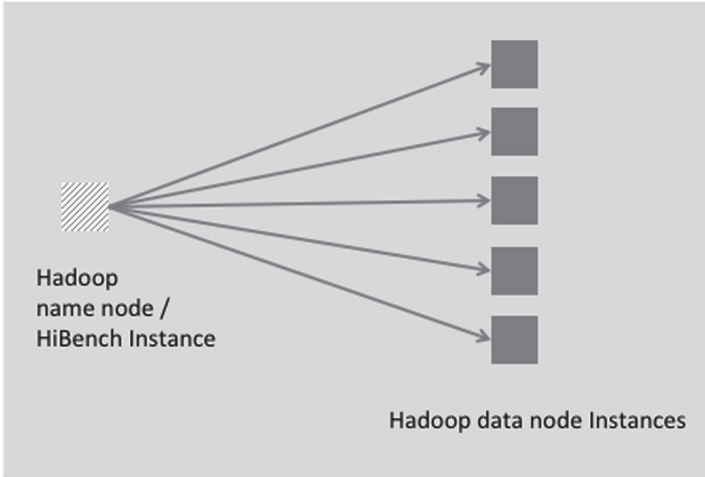


Fig. 2. K-Means application instance.

Closed division which is intended to compare hardware platforms or software frameworks and requires using the same model and optimizer as the reference implementation. The Open division is intended to foster faster models and optimizers and allows any ML approach that can reach the target quality. However, the open version still needs to use monitoring or enhanced machine learning, and one of the models is iterative improvement based on training data, simulation, or self-comparison. Therefore, the open version of MLPerf is more a comparison of machine learning algorithms, with the purpose of optimizing artificial intelligence algorithms. This paper adopts The Closed version, which specifies the model to be used and limits the values of super parameters such as batch size or learning rate, which is very fair for comparing hardware and software systems of different cloud platforms, and is very suitable as a workload to test the performance of machine learning algorithms running on cloud platforms. MLPerf is divided into two suites which are Training and Inference, corresponding to the two main parts of machine learning algorithms. There are five benchmarks about reasoning, seven benchmarks about training. The cloud platform performance evaluation method designed by this paper is to use lightweight object detection of seven Training benchmarks as the workload to test the performance of cloud platform in Training of artificial intelligence algorithms. MLPerf Training benchmarks are defined by a Dataset and Quality Target. Implementation models are optional, but almost every benchmark provides a reference implementation model. Training is the process of solving in machine learning algorithm, so the training process can better represent the performance of cloud platform. The more classical object detection algorithm and the use of lightweight data sets make it easier to deploy workload in the cloud platform. This paper finally adopts the Training of target detection algorithm (light weight) and the COCO

2017 data sets and realize it by `ssd-resnet34` model. The final Target Quality to be achieved is 23% mAP.

4 Cloud Platform Performance Evaluation Method Implementation

In this paper, design of the machine learning workload can be run on more than one GPU, in order to make the machine learning workload can take advantage of the cloud platform, to speed up the training process and save the time of the evaluation method, the default size of the cluster deployed by workload is 3. It means that machine learning workload will be deployed on a cluster of three instances. This paper intends to deploy the machine learning workload on the cloud platform using the Horovod [10] distributed framework. Horovod is a distributed machine learning training framework compatible with mainstream computing frameworks, mainly based on ring AllReduce algorithm released by Baidu in 2017. Using Horovod as a distributed framework can improve the efficiency of machine learning training algorithm and make full use of the performance of cloud platform, thus reducing the test time of the performance evaluation method designed in this paper. Choose the lightweight object detection training of MLPerf as the machine learning workload. To enable the machine learning workload to be automatically scheduled like the two workloads of the SPEC Cloud itself and automatically deployed in the Cloud platform, it needs to call the API interface of the CBTOOL to schedule the cloud platform [9]. The images of workloads which includes the machine learning workload designed by this paper must be prepared firstly before we run The SPEC Cloud Benchmark. We can prepare the first two images according to the UserGuide of the SPEC Cloud benchmark. Then we need to prepare the images of machine learning workload designed by this paper. The control node then needs to use the CBTOOL tool to create the instances with an image of the crafted machine learning workload. In general, the larger the number of instances that represent the larger the cluster of machine learning workloads, the shorter the test time will be. The default number of the instances designed in this paper is 3. The running of the machine learning workload starts with the creation of the instances, and the entire process from starting this workload to the completion of the training to the collection of results is automatically completed by running the script on control of the control node. Machine learning workloads are performed on multiple instances to take advantage of the performance of the cloud platform, which requires a better distributed framework to shorten the completion time of machine learning algorithms. As a result, the results of cloud platform performance evaluation can be obtained more quickly. In this paper, Horovod framework is selected and the advantages of the two papers, “Training ImageNet in 1 h” of Facebook and “Bringing HPC techniques to deep learning” of Baidu are combined in this paper. Ring-allreduce, a new gradient synchronization and weight synchronization algorithm, and data transfer based on MPI can greatly improve the performance of the cluster. After the cluster of instances

is created, the script gets the IP which is from intranet of instances and start the service about Horovod. The training process will run as a loop on the cluster of instances. The control node collects the results after the training ends to obtain the running time of the entire cluster of machine learning workload as a measure of the performance of the cloud platform in terms of AI algorithms. If we do not make changes to the SPEC Cloud project to create a mirror, many errors will be reported because the aarch64 architecture is not supported. To make the cloud platform performance evaluation method designed in this paper support Arm architecture, it is necessary to adapt CBTOOL to the Arm architecture. We add the following code in `get_linux_distro()` function in `osgcloud/lib/auxiliary/dependencies.py`:

```

elif \_linux\_distro\_kind.count("Kylin"):
    \_linux\_distro\_kind = "ubuntu"
    \_linux\_distro\_name = "xenial"

```

The 3.0.1 version of Redis will cause CBTOOL to run incorrectly in Arm system, which can be solved by changing the version of Redis to 2.10.3. Finally, the performance evaluation method designed in this paper is successfully adapted to Arm cloud platform, so it can also support the performance evaluation of domestic cloud platform.

5 Results and Discussion

This chapter is mainly about the experimental results of the simulation experiment on the domestic Arm cloud platform, and the analysis of the results. And this paper tested the performance evaluation method of the cloud platform designed in this paper on the cloud platform built by TaiShan 2280 v2 server and the cloud platform built by Great Wall optimus D F720 Feiteng server. The test results are as follows (Tables 1, 2, 3, 4, 5, 6, 7 and 8):

Table 1. Cloud configuration

Cloud vendor	Peng cheng laboratory
Cloud type/SUT type	Private/white box
Hardware platform	Arm64
Hypervisor	KylinVM
Cloud infrastructure	Kylin cloud

Table 2. Hardware configuration

Parameter	Value
Server	TaiShan 2280 v2
Cores num	48 cores
CPU version	Hi1620
Memory	512 G
Disk size	446 GB SSD + 4TB SATA

Table 3. Instance configuration

Parameter	Value
CPU	8
Memory	8 GB
Disk size	80 GB

Table 4. Software configuration

Parameter	Value
Python version	2.7.12
JVM version	1.7.0_95
Hadoop version	2.7.5
Cassandra version	2.1.18

Table 5. Baseline summary results for YCSB

	Throughput (ops/s)	Insert latency 99% (ms)	Read latency 99% (ms)	AI provisioning time (s)
Average	26621.3	1.947	6.897	152.4

Table 6. Baseline summary results for KMeans

	Completion time (s)	AI provisioning time (s)
Average	152.88	176.0

Table 7. Scale-out summary results for YCSB for 5 valid AIs

Av. throughput (ops/s)	Av. insert latency 99% (ms)	Av. read latency 99% (ms)	Av. AI provisioning time (s)	Performance score	Relative scalability (%)
Average	3.860	6.338	197.4	3.57	73.44

Table 8. Scale-out summary results for KMeans for 7 valid AIs

Av. completion time (s)	Av. AI provisioning time (s)	Performance score	Read relative scalability (%)
152.7	189.7	5.82	93.41

And the machine learning workload takes 330350 s to accomplish the specified precision using three instances in the cloud. So, we can get the performance of the cloud platform in terms of CPU and GPU. The entire process is controlled and automated by a cbharness to run the load and then generate results, helping enterprise users objectively evaluate the performance of the cloud platform and make reasonable choices. The performance evaluation method designed in this paper can also support the domestic cloud platform and the above results are tested on the domestic Kylin Cloud.

6 Conclusion

The traditional approach to evaluating cloud platform performance is to test the performance of the cloud platform in terms of CPU performance, such as read-write, computing, and ability of scaling the cloud platform itself. MLPerf is the most representative benchmark that can test the performance of machine learning algorithms in the cloud platform. There is no method to test the performance of cloud platform automatically and comprehensively. Therefore, this paper intends to design and implement a cloud platform performance evaluation method based on SPEC and MLPerf. After testing on Kylin cloud platform with domestic Arm architecture, the method designed in this paper has been proved to be able to run automatically and obtain more comprehensive performance test data of cloud platform. The shortage is that only time is selected as the metric of the machine learning load. Although it is intuitive and effective, it cannot reflect the performance of cloud platform in machine learning algorithm in a more detailed way. What we can do in the future work is to design new metrics, such as the PR curve with more detailed accuracy and recall rate.

References

1. Yan, Z.: White Paper on Cloud Computing and Artificial Intelligence Industry Applications. Institute of Internet Industry, Tsinghua University (2018)
2. Deloitte artificial intelligence: from expert-only to everywhere. <https://www.deloitte.co.uk/tmtpredictions/predictions/artificial-intelligence/>
3. Google. PerfKit benchmarker. <https://github.com/GoogleCloudPlatform/PerfKitBenchmarker>
4. Standard Performance Evaluation Corporation. <http://spec.org/>
5. TPC: Transaction Processing Performance Council

6. Ficco, M., Rak, M., Venticinque, S., et al.: Cloud evaluation: benchmarking and monitoring. In: Quantitative Assessments of Distributed Systems (2015)
7. MLPerf. <https://mlperf.org>
8. Silva, M., Hines, M.R., Gallo, D., et al.: CloudBench: experiment automation for cloud environments. In: ACM Symposium on Cloud Computing. ACM (2012)
9. Cloud rapid experimentation and analysis toolkit. <https://github.com/ibmcb/cbtool>
10. Sergeev, A., Del Balso, M.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint [arXiv:1802.05799](https://arxiv.org/abs/1802.05799) (2018)



Parallelization and Optimization of Large-Scale CFD Simulations on Sunway TaihuLight System

Hao Yue , Liang Deng, Dehong Meng , Yuntao Wang, and Yan Sun

Computational Aerodynamics Institute, China Aerodynamics Research and Development Center, Mianyang, China
haoeryue@hotmail.com

Abstract. TRIP is an in-house Computational Fluid Dynamics (CFD) software that can simulate subsonic, transonic, and supersonic flows with complex geometries. With the increase of computation and memory requirement for large-scale CFD simulations, it is an inevitable trend to use massively parallel computers for parallel computing. In this paper, with a dual-level hybrid and heterogeneous programming model using MPI + OpenACC, we port and optimize the TRIP software on the Sunway TaihuLight supercomputer. A series of optimization techniques, including data reconstruction, data packing, loop refactoring and array swapping, are explored. In addition, a grid preprocessing tool is developed for reducing the load imbalance caused by the non-cube shape of sub grids. Scalability tests show that TRIP can achieve 66.9% parallel efficiency of strong scaling and 96% efficiency of weak scaling when the cores are increased from 10,400 to 665,600.

Keywords: CFD · TRIP · Parallelization · Optimization · Sunway TaihuLight · OpenACC

1 Introduction

Computational Fluid Dynamics (CFD) is a traditional application in the field of high performance computing, and its high-efficiency has greatly reduced the cycles for developing aerospace vehicles. TRIP, as the first domestic CFD platform with completely independent copyright, has been successfully promoted to many research institutions which dedicated to the development of aircrafts and weapons [1]. It has almost covered all the functions of the other mainstream structural grid CFD software, and has participated in a sequence of tasks of verification and validation, such as DPW, HiLift series and other standard models. From low to supersonic flow, TRIP has shown its outstanding performance compared with the similar software in this field. So, it is widely used for the real-world CFD simulation, such as conventional aerodynamic characteristics analysis, integrative calculation of internal and external flow, aerodynamic noise and aeroelasticity problems, etc. However, with the increasing of accuracy requirements, the grid must be designed more and more finely. Therefore, it is of crucial importance to develop highly scalable tools for extreme-scale CFD simulations on state-of-the-art computing platforms.

As the first super computer system with a peak performance of over 100 Pflops [2], Sunway TaihuLight provides a favorable condition for the large-scale parallel algorithm of CFD. Constrained by factors such as power consumption, many-core heterogeneous processors have gradually become the mainstream architecture of high-performance computing. For many applications, the traditional hybrid MPI and OpenMP parallel model cannot satisfy the requirements of new computing models anymore. Therefore, various studies have been conducted on large-scale parallel program optimization for the Sunway TaihuLight and achieved a series of success [3–5]. In recent years, some efforts have been made to port CFD software to many-core systems. Such as GKUFS [6] and SWLBM, both are based on Boltzmann equation and use particle velocity distribution function as the physical quantity, have achieved great adaptability on many-core platform. These types of software can get good performance are due to the complete data independence and the choice of explicit time method. There are also some CFD software based on solving the N-S equations have been ported Sunway many-core system. Such as the direct numerical simulation software OpenCFD, which is developed by the Institute of Mechanics, Chinese Academy of Sciences, and achieved great scaling due to its high calculation memory ratio and calculation communication Ratio [7]. Another open code OPENFORM has been ported to TaihuLight, in their study, the mixed-language application was designed to overcome the compilation incompatibility problem and the speedup of $184.9\times$ on 256 CGs was achieved [8]. However, in many studies, they used designed dataset or regular grid with single block in order to suit for the architecture.

Compared with homogeneous system, there are two main challenges for us to port TRIP on TaihuLight. Firstly, unlike the traditional x86 processor, the design of the CPEs does not contain a cache, but a 64 KB user-controlled Scratch Pad Memory (SPM), which means it is vital to manage the local data memory (LDM) carefully in order to reduce the severe performance degradation caused by the frequent accesses to main memory. Secondly, due to the complex shape, we can't always design the regular mesh with single zone during the pre-processing period. So, it is inevitable that unwished non-cube sub-grids appear when splitting the original grid, which will increase the load imbalance.

In order to tackle above challenges, targeted parallel strategies were exploited for kernel functions with different degrees of data dependency in our work, such as grid-node, grid-line and grid-plane level. And a series of optimization methods such as data reconstruction, data packing, loop refactoring and array transposition were used to achieve effective many-core parallelism. As for the potential load imbalance of large-scale, an automatic pre-processing tool for rotating local sub-grid coordinate was proposed. The result shows that the kernel function of flux computation can achieve up to about $28.85\times$ speedup on a single Core-Group (CG), and the kernel function of implicit time method LU-SGS can achieve up to $3\times$ speedup. After using the automatic local coordinate tool, a significant improvement of parallel efficiency can achieve.

2 Background

2.1 Sunway TaihuLight Architecture

The Sunway TaihuLight supercomputer provides a theoretical peak performance of 125PFlops. Each computer node contains a SW26010 heterogeneous many-core processor. The architecture of the Sunway processor is shown in Fig. 1, which is composed with 4 CGs. Each CG contains a Management Processing Element (MPE, or Host) and 64 Computing Processing Elements (CPE, or Device).

In terms of the memory architecture, each CG has 8 GB main memory, which makes up the 32 GB main memory in a SW26010 processor. Each MPE has a 32-KB L1 data cache and a 256-KB L2 instruction/data cache, and each CPE has its own 16-KB L1 instruction cache and a 64-KB LDM (also known as SPM as a user-controlling fast buffer). The CPE has two types of memory access to the main memory: DMA and Global load (Gld). According to a Stream benchmark test, when being accessed by Gld instructions, the maximum bandwidths are only 3.88 GB/s. Correspondingly, when using DMA mode, the maximum Copy bandwidth reaches 27.9 GB/s [2].

2.2 The Euler Equations

For simplicity, the Euler equation [9] is used in following discussion. The Euler governing equation in generalized curvilinear coordinate system is expressed

$$\frac{\partial \tilde{Q}}{\partial \tau} + \frac{\partial \tilde{F}}{\partial \xi} + \frac{\partial \tilde{G}}{\partial \eta} + \frac{\partial \tilde{H}}{\partial \zeta} = 0 \tag{1}$$

Where, \tilde{Q} is the conservation variable in generalized curvilinear coordinate system, \tilde{F}, \tilde{G} and \tilde{H} are the convection flux vectors in the generalized curvilinear coordinate, τ, ξ, η and ζ correspond to t, x, y and z in the Cartesian coordinate, respectively.

Taking a unit cell whose center is denoted by (i, j, k) in the generalized curvilinear coordinate system, we have a semi-discretized system in the form of

$$\left(\frac{\partial \tilde{Q}}{\partial \tau} \right)_{i,j,k} = - \left(\tilde{F}_{i+\frac{1}{2},j,k} - \tilde{F}_{i-\frac{1}{2},j,k} + \tilde{G}_{i,j+\frac{1}{2},k} - \tilde{G}_{i,j-\frac{1}{2},k} + \tilde{H}_{i,j,k+\frac{1}{2}} - \tilde{H}_{i,j,k-\frac{1}{2}} \right) \tag{2}$$

In order to simplify the expression, here $\Delta \zeta, \Delta \eta, \Delta \xi$ is 1.

2.3 Spatially Discretization

Define the spatially discretization (right item of Eq. (2)) as RHS, and it refers to the process of convection flux in the Euler equation. In this paper, a third-order upwind scheme based on Roe’s approximate Riemann solver was used. For simplicity, we just discuss the x component, and the other components are similar. It can be expressed

$$\tilde{F}_{i+1/2} = 1/2 \left[F(Q_{i+1/2}^L) + F(Q_{i+1/2}^R) \right] - 1/2 \left| \tilde{A} \left(Q_{i+1/2}^L, Q_{i+1/2}^R \right) \right| (Q_{i+1/2}^R - Q_{i+1/2}^L) \tag{3}$$

Where, $Q_{i+1/2}^L$ and $Q_{i+1/2}^R$ are interpolated by the several left and right adjacent nodes along i -dimension respectively, and $\tilde{A}(Q_{i+1/2}^L, Q_{i+1/2}^R)$ is the matrix of Roe average. In particular, the third-order precision Muscl scheme is used to interpolate $Q_{i+1/2}^L$ and $Q_{i+1/2}^R$, its form is

$$Q_{i+1/2}^L = g^L(Q_{i-1}, Q_i, Q_{i+1}); \quad Q_{i+1/2}^R = g^L(Q_i, Q_{i+1}, Q_{i+2}) \quad (4)$$

2.4 Time Discretization

After discretizing and linearizing the time item of Eq. (2), we can get a large-scale system of linear equations just like

$$Ax = b \quad (5)$$

Where, A denotes LHS matrix, b denotes RHS matrix and x denotes solution vector. When using LU-SGS scheme to solve Eq. (5), matrix A can be decomposed as

$$A = D + L + U \approx (D + L)D^{-1}(D + U) \quad (6)$$

Where D is the diagonal vector, L is a lower triangular matrix and U is an upper triangular matrix. So Eq. (5) can be expressed

$$(D + L)D^{-1}(D + U)x = b \quad (7)$$

Using a temporary variable y , the Eq. (7) can be reconstructed to two symmetrical Gauss-Seidel functions as Eq. (8) shown. The former function is for forward updating and the latter for backward updating.

$$(D + L)y = b; \quad (D + U)x = Dy \quad (8)$$

Finally, the solution can be gotten by using the back-generation method.

2.5 The Workflow of TRIP

The algorithm flow chart is illustrated in Fig. 2. The first step is the initialization of flow field, which specifies the cell number of grids, the cell coordinates, and the cell values. Then, the kernel iteration process of the algorithm LU-SGS is executed until the results are converged. When running, the flux computation of RHS accounts for 46.3% of entire time, implicit time scheme LU-SGS accounts for 37.75% and other items accounts for 15.94%.

When using parallel mode, TRIP adopts static data assignment method, which means that the original grid must be decomposed into multiple sub-grids before running. So, an in-house tool *Trip_mbsplit* is used, and the tool will produce three files for each process respectively, including grid coordinate file (*.grd), boundary information file (*.inp) and communication file (*.msg). When running, each process reads its own information from corresponding sub-grid files separately, then runs step by step according to the flow in Fig. 2. In order to exchange the boundary information of sub-grids, the MPI point-to-point communication mechanism is used. Finally, after all processes finished, the MPI collective communication is used to merge the flow field and integrate the aerodynamic force.

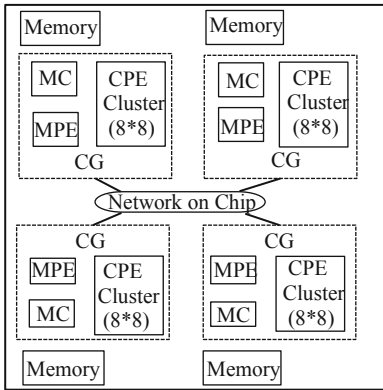


Fig. 1. General architecture of the SW26010 process.

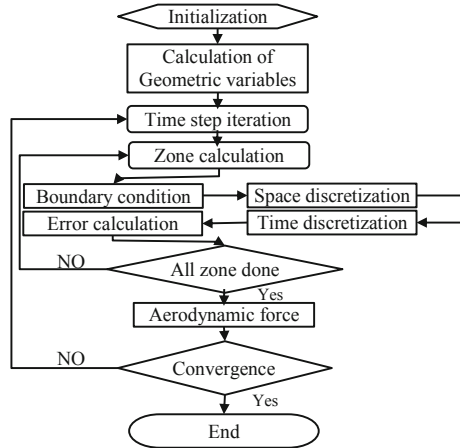


Fig. 2. TRIP workflow chart

3 Parallelization Strategies

In order to port TRIP to Sunway TaihuLight, a hierarchical domain partition strategy for parallelization is employed. At the process level, the whole computational domain is decomposed to multiple subdomains, with each assigned into a MPI process and running on a single CG. In particularly, the boundary information is exchanged among processes before updating the flow field in each subdomain. At a CG level, according to the difference of data dependency in kernel functions, corresponding parallel strategies based on OpenACC are exploited. In what follows, the specific strategies are discussed.

3.1 Parallelization of Kernel Functions with Data Independence

The functions with completely independent data structure are the easiest parts to achieve many-core parallelization. Such as the function used to converse the conservation variables into original variables, and the function used to correct the pressure density, etc. For these types of kernel functions, the data can be transferred to CPEs directly according to the limited LDM, and implement parallelization at the grid-node level easily, as shown in Fig. 3 (left). If the calculation of every node is relatively simple in each step, a strategy with grid-line level or grid-plane level can be used to guarantee sufficient computation. As shown in Fig. 3 (middle) and Fig. 3 (right), each CPE solves a line or plane of grid respectively. Since the threads and data are scheduled in the smallest unit, great speedup and bandwidth performance can be obtained.

3.2 Parallelization of Kernel Functions with Weak Data Dependency

For some kernel functions, the variables updating must depend on the data of their adjacent nodes, such as rhs_ch_i , rhs_ch_j and rhs_ch_k and $rabc$, etc. Specifically, rhs_ch_i , rhs_ch_j and rhs_ch_k are used to calculate the convection flux along x ,

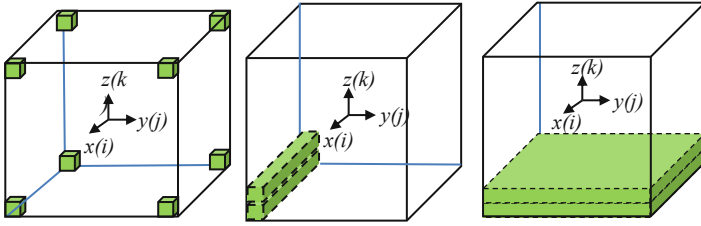


Fig. 3. Parallelization strategy of different level (Figure from left to right refers to strategy based on grid-node, grid-line, and grid-plane respectively)

y and z dimension respectively, and $rabc$ is used to calculate the spectral radius of coefficient matrices in the subroutine LU-SGS. Although these calculations depend on their neighbors, they will not affect the result as long as the relative data is copied to the LDM in advance. In particular, the reason why the flux calculation is divided into three one-dimensional functions along x , y and z component is for minimizing the duplication of redundant data. For example, when calculating the flux $\tilde{F}_{i+1/2,j,k}$ through the surface $ABCD$, as shown in Fig. 4 (left), variables at cell center $(i - 1, j, k)$, (i, j, k) , $(i + 1, j, k)$ and $(i + 2, j, k)$ are requested according to the algorithm mentioned in Sect. 2.3. In order to take advantage of the continuity of the array in low dimensions, the grid-line parallelization scheme was mainly used. As shown in Fig. 4 (right), the tasks were scheduled in the outer j and k dimensions and the cells of i dimension were calculated sequentially by each CPE. Here, $batch_size$ refers to the number of cells can be copied once, and it will be discussed later to maximize the sustained performance.

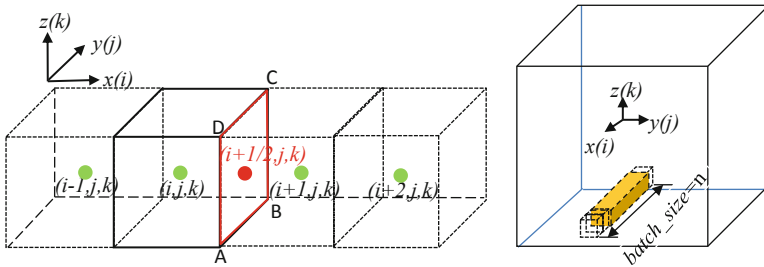


Fig. 4. Parallelization strategy for weak data-dependent functions (Left illustration shows the dependence of flux computation and the right is the corresponding grid-line parallelization)

3.3 Parallelization of Kernel Functions with Strong Data Dependency

Calculation with strong data dependency mainly exists in the implicit time scheme LU-SGS. As shown in Fig. 5 (left), during the forward (lower triangular matrix) updating process of LU-SGS, the updating node (i, j, k) depends on its adjacent updated nodes $(i - 1, j, k)$, $(i, j - 1, k)$ and $(i, j, k - 1)$. On the contrary, the nodes $(i + 1, j, k)$, $(i, j + 1, k)$ and $(i, j, k + 1)$ are needed to update the node (i, j, k) during the backward (upper triangular

matrix) updating. In this paper, a parallel algorithm based on diagonal-hyperplane was used to implement thread-level parallelization for LU-SGS. As shown in Fig. 5 (right), the strategy mainly utilizes the independence of the nodes on the diagonal-hyperplane to implement the many-core parallelization. In other words, the nodes on the hyperplane can be calculated concurrently by 64 CPEs, then, the whole domain is calculated plane by plane. In particular, the subscripts i, j, k on the diagonal-hyperplane add up to a constant. So, it is easy to find their association based on this character.

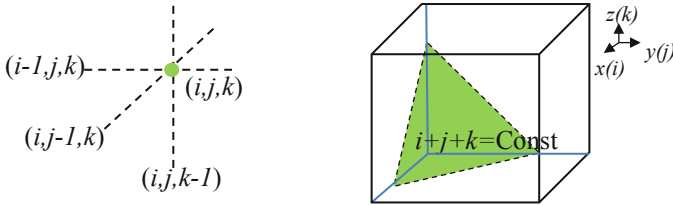


Fig. 5. Parallelization strategy for LU-SGS (Left illustration shows the dependence of LU-SGS and the right shows the corresponding diagonal-hyperplane parallelization strategy)

4 Optimization Within a CG

4.1 Refactoring the Loop Structure

TRIP is developed based on Fortran language, and most of data-structures (arrays) are four-dimensional arrays with the layout of (n, i, j, k) . Such as array $flw(mfl, i, j, k)$, which stores ρ, u, v, w, p respectively when mfl takes 1 to 5. The elements of array in Fortran are stored in column-major order, so, it is best to access array from low to high dimension. In original code, there are many non-compact sub-loops nested in the main loop of RHS, and these sub-loops contain big data access. As shown in Algorithm 1, there are four main parts to update the flux in k -dimension, which are copying array, calculating $\tilde{H}_{i,j,k+1/2}$, calculating $\Delta\tilde{H}$ and updating rhs . And the second one is the most time consuming part. Since the start and end index of k -dimension in the sub-loops are different, the inner-layer loop cannot be merged into a whole loop, so that the entire memory access is not continuous. To achieve continuous memory access, the original loop is refactored to two continuous four-dimensional loops reasonably, as shown in Algorithm 2. Where, subroutine $f1$ is used to calculate $\tilde{H}_{i,j,k+1/2}$, and the result is saved in $fu4d(:, i, j, k)$. Another subroutine $f2$ is used to calculate $\Delta\tilde{H}$ and updating rhs . Then, it is easy to use OpenACC to implement parallelization. It must be mentioned that the relative data of adjacent nodes $(i, j, k - 1), (i, j, k), (i, j, k + 1)$ and $(i, j, k + 2)$ needs to be transferred into subroutines $f1$ or $f2$ due to the data dependency.

Algorithm1 subroutine rhs_ch_k

```

Require:4D-Arrays,cnk, flw,fu2d,rhs...
1:for j ← js to je do
2:  for i ← is to ie do
3:    for k ← ks-1 to ke+2 do
4:      cn1d(:,k) = cnk(:,i,j,k)
5:      q1d(:,k) = flw(:,i,j,k)
6:    end for
7:    for k ← ks-1 to ke do
8:      calculate  $\tilde{H}_{i,j,k+1/2}$ , saved in fu2d(:,)
9:    end for
10:   for k ← ke to ks do
11:     calculate  $\tilde{H}_{i,j,k+1/2} - \tilde{H}_{i,j,k-1/2}$ 
12:   end for
13:   for k ← ks to ke do
14:     update rhs
15:   end for
16: end for
17: end for

```

Algorithm2 subroutine rhs_ch_k'

```

Require:4D-Arrays,cnk, flw,fu4d,rhs...
1: !$acc parallel loop collapse(2) local() copy()
2: for k ← ks-1 to ke do
3:  for j ← js to je do
4:    call f1(cnk(:,j,k),flw(:,j,k-1),flw(:,j,k), &
5:    & flw(:,j,k+1),jflw(:,j,k+2),fu4d(:,j,k)...)
6:  end for
7: end for
8: !$acc end parallel loop
9: !$acc parallel loop collapse(2) local() copy()
10: for k ← ke to ks
11:  for j ← js to je
12:    call f2(fu4d(:,j,k),fu4d(:,j,k-1),rhs(:,j,k))
13:  end for
14: end for
15: !$acc end parallel loop

```

4.2 Data Transfer and Batch_Size Determination

As for the reason why the subroutines *f1* and *f2* are created is for transferring data more accurately. In fact, the data like *flw*, *cnk*, *rhs* are all organized as pointers type, and sometimes, the data requirement is complex when transferring data to LDM. So, it is difficult to make right analysis automatically for compiler by just using simple *acc copy* directive at outer loop. Therefore, when adopting this method, it can be more explicit and more accurate to transfer data by using *acc data copy* directive in the subroutine *f1* and *f2*.

Due to the limited LDM, the whole data of *i*-dimension cannot be transferred into LDM at one time. So, a further decomposition for *i*-dimension needs to be done. Assume *batch_size* is the amount of data can be transferred to LDM at one time, in order to maximize the sustained performance of LDM, the *batch_size* can be estimated by Eq. (9). Where, *Var_nneed* refers to the amount of requirement data requirement in a single step.

$$\text{Var}_n\text{need} \times \text{batch_size} \times 8 \text{Byte} \leq 64 \text{KB} \times 1,024 \quad (9)$$

4.3 Other Optimizations

Array Swapping. For some arrays with discontinuous access, the *swapin* clause is used in order to avoid inefficient memory access. For example, when transferring array *btem*(*i*, *j*, *k*, *n*) to LDM, the loop order is *n*, *i*, *j*, *k* from inside to outside, it is easy to guarantee the continuous access by using clause *swapin(btem(dimension order: 4, 1, 2, 3))*.

Data Pre-sorting. When using the diagonal-hyperplane parallel strategy for LU-SGS parallelism, the data required on the hyperplane is not continuous. So, they cannot be transferred into LDM directly because the data chunk is not regular. In order to avoid massive discrete access of CPEs, the data on the hyperplane is sorted in advance on the host. Its essence is to allocate a new array for storing sorted data.

Data Packing. For data transmission, the transmission efficiency of scattered variables is worse than the efficiency of batch data copying, so packing the scattered variables into one large variable can reduce the cost of data copying. In this paper, the *packin* clause is used to improve the data transmission efficiency.

5 Load Balancing for Large-Scale Parallelization

5.1 Load Balancing Analysis

In large-scale CFD parallel computing, load balancing is one of the key factors to scale application efficiently. Typically, for the heterogeneous systems, the load balance needs to be measured from three aspects, which are computing balance, communication balance and communication calculation ratio, for which three quantitative indicators were specified [6, 10], as shown in Eq. (10–12).

$$\sigma_v = \frac{V_{\max} - V_{ave}}{V_{ave}} \times 100\% \quad (10)$$

$$\sigma_s = \frac{S_{\max} - S_{ave}}{S_{ave}} \times 100\% \quad (11)$$

$$r_i = S_i/V_i \times 100\% \quad (12)$$

Where, σ_v , σ_s and r_i refer to factor of volume balance, area balance and area volume ratio, respectively. V_{\max} and V_{ave} refer to the max and average volume among sub-grids respectively. S_{\max} and S_{ave} refer to the max and average area among sub-grids. S_i and V_i refer to the area and volume of the i -th sub-grid. According to these three factors, it is easy to find that the sub-grids with a close cube shape are best. Because a cube-shaped sub-grid has the advantage of smallest communication calculation ratio. However, in a real case, especially for large-scale parallel computing with complex shapes, it is inevitable to produce non-cube sub-grid due to the limit of complex physical conditions or other limits. Compared to other system, it is more severe when using the grid-line parallelization strategy in this paper.

Assume the amount of nodes in a sub-grid is $size_i$, $size_j$ and $size_k$ along i , j , k -dimension respectively, and the maximum capacity of LDM at each time is $batch_size$ in a kernel function. Then, the actual amount can be transmitted to LDM once will be:

$$block_size = \begin{cases} size_i, & batch_size \geq size_i \\ batch_size, & batch_size < size_i \end{cases} \quad (13)$$

Where, $block_size$ refers to the actual amount of nodes can be transferred to LDM at one time. So, when finished transferring the entire sub-grid, the number of DMA transmission (N_{DMA}) can be calculated via the Eq. (14)

$$N_{DMA} = size_j \times size_k / 64 \times size_i / block_size \quad (14)$$

As shown in Fig. 6 (left), assuming the size of sub-grid A is $8 \times 200 \times 100$, and the *batch_size* can be 120 for a kernel function according to the Eq. (9). Nevertheless, the sub-grid has only 8 grid nodes along the *i*-dimension, so that the actual *block_size* can only be 8 according to Eq. (13), and the number of DMA transmission is 313 according to Eq. (14). For another sub-grid B, which is the same volume as A but its size is $200 \times 100 \times 8$, as shown in Fig. 6 (right), according to above equations, only 26 DMA transmissions are required.

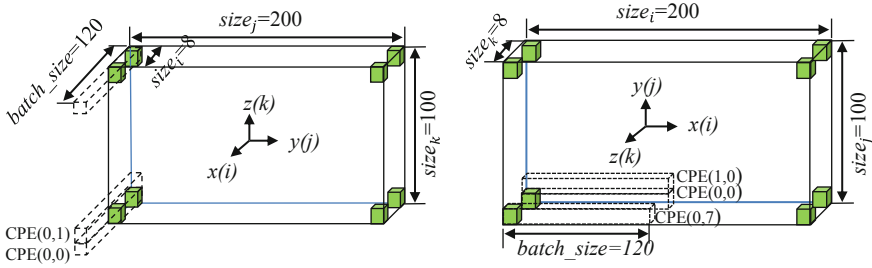


Fig. 6. Illustration of data transmission for different sub-grids (Left illustration is sub-grid A with size $8 \times 200 \times 100$, the right is sub-grid B with size $200 \times 100 \times 8$)

In order to measure the deviation degree between the $size_i$ of sub-grid n with $size_i$ of the same volume cube sub-grid, the index σ_i is defined, as shown in Eq. (15).

$$\sigma_i = size_i / \sqrt[3]{V_n} \quad (15)$$

Where, V_n refers to the volume of sub-grid n and $size_i$ refers to the amount of grid nodes along *i*-dimension of this sub-grid. According to the analysis above, when the parallel tasks for outer loop are sufficient, for two sub-grids with same volume but different σ_i , the sub-grid with bigger σ_i will be better because its N_{DMA} is much less. Since the number of CPEs within a CG is only 64, the parallel tasks of *j* and *k* dimension are sufficient for most situations. So, for the sub-grid with smaller σ_i just like A shown in Fig. 6 (left), if it could be transformed into B as shown in Fig. 6 (right), the load-balance would be better.

5.2 Improvement of Load Balancing

In the original partitioning algorithm of *Trip_mbsplit*, the influence of σ_v , σ_s and r_i has been taken into consideration, so a close cube-shaped sub-grid is prior. However, if it was inevitable to produce some sub-grids like A-shaped due to the special limits, in order to reduce load imbalance but still retain the advantages of original partitioning algorithm, a grid preprocessing tool *Trip_trans* for optimizing A-shaped sub-grids was developed.

When using *Trip_mbsplit* tool for static decomposition, the first step is dividing the flow field into multiple sub-region, and then establishing a local coordinate *I-J-K* system in each region, after that, each process starts calculating its local field independently. Since the process of reading and calculating of the sub-region is independent among

the processes, it is possible to convert A-shaped sub-grid into B-shaped sub-grid shown in Fig. 6 just by changing its local coordinate system. Before TRIP starts running, *Trip_trans* will check all the sub-grids divided by *Trip_mbsplit* and filter out the sub-grids whose σ_i is less than 1, then, it will adjust these sub-grid’s local coordinate to ensure *i*-dimension contains the most nodes. In particularly, the key to use the tool is to deal with the problem of information exchange between the processes which contain share mating surfaces, which means that the tool will not only adjust the file *.grd, but also adjust its corresponding boundary file *.inp and communication file *.msg.

6 Evaluation

6.1 Evaluation Within a CG

In order to evaluate the optimizations in a single CG, three small cases with same volume but different shapes were set up, which were $160 \times 160 \times 160$, $1,600 \times 160 \times 16$, and $16 \times 160 \times 1,600$ respectively. Since the runtime of kernel functions with complete data independency is relatively less, detailed tests and analysis of these types of functions are not showed here. While, the performance of flux computation and LU-SGS are mainly talked about at following analysis.

Figure 7 shows the optimization result of flux calculation within single CG, where, *Host_v0* represents the benchmark on the host, *Host_v1* represents the optimized version1 on the host by refactoring loop structure and other methods, and *OpenACC* represents the optimized parallelization version accelerated by CPEs. Firstly, it can be seen that the non-cube grid takes longer time than cube grid. After host optimization, the runtime of flux computation is almost same, which eliminates the load imbalance at host level. Secondly, compared with *Host_v1*, the magnitude of speedup by using *OpenACC* is consistent with the magnitude of σ_i , which verifies the conclusion mentioned in Sect. 5.1, that is to say, the larger the value of σ_i , the greater the acceleration of *OpenACC*.

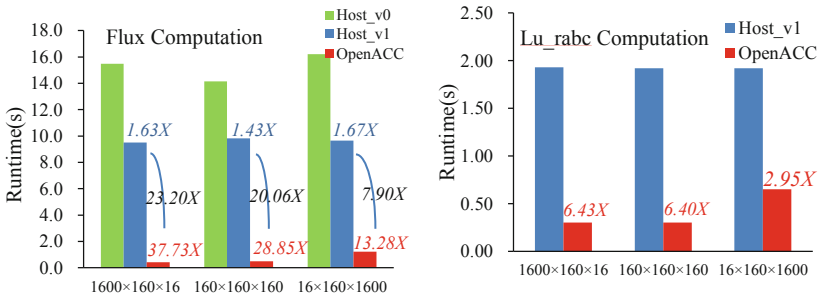


Fig. 7. Optimization result of kernel functions with weak data dependency (The left illustration shows speedup of flux computation, the right shows speedup of lu_rabc)

The LU-SGS method mainly contains three kernel functions, which are *lu_rabc*, *lu_lowr* and *lu_upper*. Where, the first one contains weak data dependence but the latter two contain strong data dependence. As shown in Fig. 7 (right), since *lu_rabc* is not decomposed into one-dimension like flux, it needs more adjacent nodes, its speedup is only $6.40\times$ when the grid size is $160 \times 160 \times 160$, which is less than the speedup of flux computation. But Similarly, the speedup of these three cases are consistent with their σ_i .

For the kernel functions of *lu_lowr* and *lu_upper*, although data pre-sorting mentioned in Sect. 4.3 can reduce the Gld access of CPEs, it needs redundant calculation to prepare data on the host. Therefore, the parallel efficiency is reduced inevitably. For grid with size of $160 \times 160 \times 160$, although the speedup is more than $17\times$ in the actual calculation part, the final speedup only achieves about $3\times$ because the data pre-sorting and recover process occupies most of time.

Although for grids with same volume, the grid with larger σ_i can obtain better speedup, it is still worse than a close cube-shaped grid, because it will take longer time in the boundary calculation, as shown in Fig. 8. Further, if the non-cube grid involves inter-communication with other process in a large-scale test, correspondingly, it will take longer time due to its larger halo area. Therefore, the result shows that it is undesirable to divide all sub-grids into strip-shaped blindly. That also verifies the rationality that we just use *Trip_trans* tool to adjust the non-cube grid produced under the inevitable situation but without changing the original cube-prior partition algorithm.

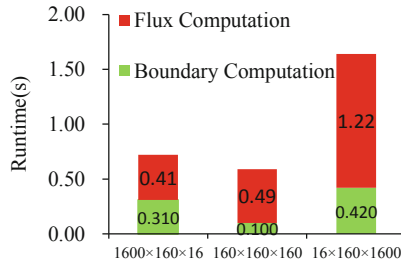


Fig. 8. Optimization results of flux and boundary control computation

6.2 Evaluation of Large-Scale Parallelization

In this paper, a large-scale test is performed based on the real AGARD 445.6 wing model. The model's aspect ratio is 1.6525, the taper is 1.5207, and the quarter-chord angle is 45° . The original coarse grid consists of 70 zones, and the total number of grid cells is 3,207,936. Refining the grid through our in-house software to generate a series of finer grids for scalability test.

Strong Scaling Test. Refining the original grid along i , j , k -directions by $7\times$ respectively in order to generate the finer grid with a total cells number of 1,100,322,048 for strong scalability test. Specifically, two comparative tests were set in order to verify the

strong scaling with processes increased from 160 to 10,240 by a factor of 2. Where, one was adjusted by *Trip_trans* and another one wasn't. As shown in Fig. 9, when adjusting sub-grids by *Trip_trans*, the efficiency has been greatly improved compared with before, especially for scale increased from 1,280 to 5,120 processes, the parallel efficiency has been improved about 20% to 30%. In particularly, the most time-consuming process of each scale is picked up, as shown in Fig. 9 (right), obviously, the final efficiency is mainly limited by the slowest process. In order to show the reason why these processes are slow to calculate, their corresponding sub-grid information is listed, as shown in Table 1. Where, version0 represents the information before adjustment by *Trip_trans* and version1 represents the one after optimization. Obviously, before the adjustment, although the volume of each sub-grid is smaller than the average volume of all sub-grids, the running speed is still the slowest since σ_i is very small. While, after the adjustment, the slowest process is no longer dominated by σ_i , but depend on the actual larger grid volume of the sub-grid. In addition, the parallel efficiency can stay above 80% when 332,800 cores are used, but drop to 66.9% when 665,600 cores are used. This is expected because, as more processes are used, the subdomain size decreases and the ratio of communication to computation becomes large and eventually hinders the overall performance.

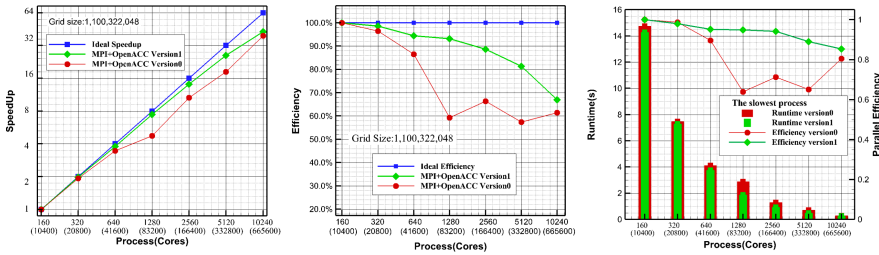
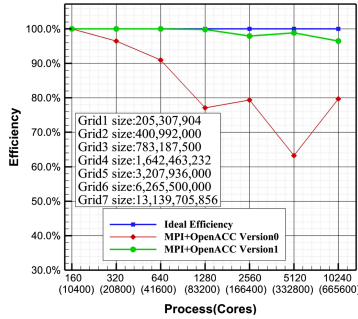


Fig. 9. Result of strong scaling test (The left and middle illustration are speedup and efficiency of hybrid MPI + OpenACC respectively, the right illustration shows runtime of slowest process at each scale. Where *Version0* represents the result before adjusting sub-grids by using *Trip_trans*, *Version1* represents the result after adjusting)

Weak Scaling Test. The AGARD445.6 wing is also used for weak scalability test. Figure 10 shows the results of the parallel efficiency of weak scalability before and after the load balance optimization. Similarly, Version0 represents original result and Version1 represents optimized one. In the test, the calculation scale is increased from 10,400 cores to 665,600 cores, and the corresponding grid magnitude is increased from 200 million to 13.1 billion. It can be clearly seen that the optimized weak scalability has been greatly improved and the parallel efficiency can reach more than 96% when 665,600 cores are used.

Table 1. Information comparison of the slowest process before and after adjustment

Process	$(V - V_{avr})/V_{avr}$ (%) (version0)	σ_i (version0)	$(V - V_{avr})/V_{avr}$ (%) (version1)	σ_i (version1)
160	-0.0004	0.292	3.4431	1.334
320	-0.0161	0.650	4.8827	1.056
640	0.1402	0.180	3.0587	1.030
1,280	-0.0194	0.080	4.8149	1.536
2,560	-0.0943	0.128	3.1283	1.103
5,120	-0.2474	0.127	4.8437	1.606
10,240	-0.0422	0.306	4.1206	2.411


Fig. 10. Result of weak scaling test (*Version0* represents the result before adjusting sub-grids by using *Trip_trans*, *Version1* represents the result after adjusting)

7 Conclusion

This paper reports our experience on porting and optimizing the TRIP software on TaihuLight. Various suitable mapping schemes such as grid-node, grid-line and grid-plane are proposed based on the degree of data dependency. Particularly, the precise data transmission schemes aiming for different kernel functions utilizes the 64K SPM effectively. By using the OpenACC directive, many kernel functions are parallelized on CPE clusters. On a $160 \times 160 \times 160$ grid, the spatially discretization module can achieve $28.85 \times$ speedup and the time discretization can achieve $3 \times$ speedup. Moreover, the matching sub-grid coordinate rotation tool *Trip_trans* can effectively reduce the impact of load imbalance caused by the difference of cell number along i , j and k dimension. Finally, a real AGARD445.6 wing model is tested, 66.9% parallel efficiency of strong scaling and 96% efficiency of weak scaling can be achieved when the cores are increased from 10,400 to 665,600. Our work may be used as a reference for other CFD applications based on structured grid when porting them to TaihuLight. However, due to the discontinuity of memory access, the parallelization of LU-SGS based on diagonal-hyperplane

strategy cannot obtain great performance, which results in the further optimization bottleneck of the whole calculation. In future work, the optimization for the implicit time algorithm still needs further exploration and implementation.

Acknowledgment. This work was supported by National Key Research and Development Program under grant# 2016YFB0200703 and National Numerical Windtunnel Project.

References

1. Yuntao, W.: Development and application of TRIP2.0_SOLVER. *Acta Aerodynamica Sinica* **106**(4), 2205 (2007)
2. Haohuan, F.: The Sunway Taihu Light supercomputer: system and applications. *Sci. China (Inf. Sci.)* **59**(07), 113–128 (2016). <https://doi.org/10.1007/s11432-016-5588-7>
3. Jian, Z., Chunbao, Z.: Extreme-scale phase field simulations of coarsening dynamics on the Sunway TaihuLight supercomputer. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City. IEEE Press (2016)
4. Chao, Y., Wei, X.: 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City. IEEE Press (2016)
5. Fu, H., He, C.: 18.9-Pflops nonlinear earthquake simulation on Sunway TaihuLight: enabling depiction of 18-hz and 8-meter scenarios. In: *International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver. IEEE Press (2017)
6. Ahusborde, E., Glockner, S.: A 2D block-structured mesh partitioner for accurate flow simulations on non-rectangular geometries. *Comput. Fluids* **43**(1), 2–13 (2011)
7. Fang, L., Zhihui, L.: Research on adaptation of CFD software based on many-core architecture of 100P domestic supercomputing system. *Comput. Sci.* **47**(01), 24–30 (2020)
8. Delong, M., Minhua, W.: Porting and optimizing OpenFOAM on Sunway TaihuLight system. *Comput. Sci.* **10**(44), 64–70 (2017)
9. Chao, Y.: *The Computational Fluid Dynamics Method and its Application*. Beihang University Press, Beijing (2006). (in Chinese)
10. Juan, Z., Linsheng, L.: Automatic partition algorithm based on multi-region and multi-code problem. *Comput. Eng.* **36**(9), 73–75 (2010)

New Trends of Technologies and Applications



Liquid State Machine Applications Mapping for NoC-Based Neuromorphic Platforms

Shiming Li, Lei Wang^(✉), Shiyong Wang, and Weixia Xu

College of Computer Science and Technology,
National University of Defense Technology, Changsha, China
{lishiming15,leiwang,wangshiyong18,xuweixia}@nudt.edu.cn

Abstract. Liquid State Machine (LSM) is one of spiking neural network (SNN) containing recurrent connections in the reservoir. Nowadays, LSM is widely deployed on a variety of neuromorphic platforms to deal with vision and audio tasks. These platforms adopt Network-on-Chips (NoC) architecture for multi-cores interconnection. However, a large communication volume stemming from the reservoir of LSM has a significant effect on the performance of the platform. In this paper, we propose an LSM mapping method by using the toolchain - SNEAP for mapping LSM to neuromorphic platforms with multi-cores, which aims to reduce the energy and latency brought by spike communication on the interconnection. The method includes two key steps: partitioning the LSM to reduce the spikes communicated between partitions, and mapping the partitions of LSM to the NoC to reduce average hop of spikes under the constraint of hardware resources. This method is also effective for large-scale of LSM. The experimental results show that our method can achieve $1.5\times$ reduction in end-to-end execution time, and reduce average energy consumption by 57% on 8×8 2D-mesh NoC and average spike latency by 23% on 4×4 2D-mesh NoC, compared to SpiNeMap.

Keywords: Mapping · Liquid State Machine · Network-on-Chip

1 Introduction

The deepening study of the neural system has driven the development of the third generation artificial neural network (ANN) - spiking neural network (SNN) [20]. Compared with the conventional ANNs, SNNs have more biological characteristics and hardware efficiency [6]. The liquid state machine (LSM) is a recurrent SNN that was proposed to mimic actual neural activations in the brain [21]. Like other SNNs, spikes are used to transmit signals between neurons of LSM. The neuron accumulates membrane potential after receiving a stimulus. If its membrane potential exceeds the firing threshold, the neuron generates a spike. LSM consist of a reservoir of neurons (“Liquid”) receiving input spike trains and a group of readout neurons receiving signals from the reservoir. Neurons in the

reservoir are randomly connected by synapses. The reservoir is a recurrent structure. Training LSM is simpler than another feedforward neural network because it only needs to train the readout layer. Because of this, LSM is currently widely deployed on a variety of neuromorphic platforms such as TrueNorth [1], Loihi [5], and so on.

In order to implement more neurons, most of the current neuromorphic platforms are multi-core system-on-chip. Each core can accommodate multiple neurons, and the cores use Network-on-Chip (NoC) to communicate data. Due to the connection in the reservoir of LSM is chaotic and sparse, LSM cannot perform efficiently with low latency and power consumption on neuromorphic platforms. The basic solution is dividing a reservoir of LSM into many partitions, and then place the partitions on each core of neuromorphic platform. During execution of basic solution, a series of hardware limitations brought by the neuromorphic platform need to be considered, such as limited cores in the neuromorphic platform, and limited capacity of each core.

Currently, there are fewer mapping methods to efficiently deploy SNNs or LSMs on NoC-based neuromorphic platforms, such as SCO [18], SpiNeMap [3], PACMAN [8], NEUTRAMS [11], etc. But these mapping methods still have some problems. SCO, PACMAN, and NEUTRAMS can only map fixed-structure and feedforward SNNs, and do not support mapping LSM networks with recurrent connections. At the same time, these three methods are not efficient in mapping. The mapping goal of SCO is to maximize the utilization of the core, so the sequential mapping method is adopted, but this method does not optimize the spikes communication between cores, resulting in increased spike latency and power consumption. PACMAN divided the SNN to reduce the number of spikes between the cores, but after the division, it was directly mapped to the NoC, so that there was a risk of contention on the NoC. Although SpiNeMap supports LSM mapping and uses a two-stage optimization method to reduce the power consumption and latency of the neuromorphic platform, the entire process will take a huge amount of time for large-scale SNNs and only find out a local optimal solution.

There are two challenges to the mapping LSM on the neuromorphic platform. First, most methods do not support LSM because of its recurrent connections. At the same time, for large-scale LSMs, the partitioning process consumes a lot of time and hard to find the best solution with the minimized spike communications among the cores of the neuromorphic platform; Second, the large-scale NoC will increase the search space and reduce the efficiency of the search algorithm during the mapping process. It takes a lot of time to find out the best mapping scheme that minimizes the spike latency and energy of the NoC-based neuromorphic platform.

In this paper, we propose a method for mapping a large-scale LSM onto an NoC-based neuromorphic platform by using the mapping toolchain - SNEAP [19]. The LSM mapping method includes three steps: profiling, partitioning, mapping. First, the profiling step will convert the reservoir of LSM with recurrent connections and spike trace into a traffic graph; then, the partitioning step

will use multi-level graph partitioning algorithm to partition the traffic graph into multiple partitions and aim to minimize spike communications among the partitions; finally, in the mapping step, the simulated annealing (SA) algorithm is used to find out a mapping scheme that places partitions on the cores of NoC-based neuromorphic platform to optimize latency and energy.

Our contributions of this paper as follows:

- We propose a LSM mapping method to map LSM to the underlying NoC-based neuromorphic platform.
- LSM mapping method can effectively reduce latency and energy on the platform, when we deploy the LSM on the NoC-based neuromorphic platform.
- We evaluate the proposed approach from a complex (i.e., latency, energy and end-to-end execution time) viewpoint and support our claims by experimental data obtained from a cycle accurate NoC simulator.

Using different size of LSMs, we show that our method can achieve $1.5\times$ reduction in end-to-end execution time, and reduce average energy consumption by 57% on 8×8 2D-mesh NoC and average spike latency by 23% on 4×4 2D-mesh NoC, compared to SpiNeMap.

2 Background and Related Works

2.1 Liquid State Machine (LSM)

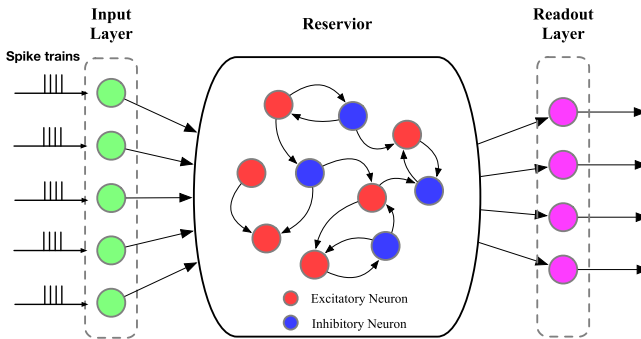


Fig. 1. The topology of LSM.

The topology of the LSM consists of three main components as shown in Fig. 1, which are input layer, reservoir and readout layer respectively. The reservoir in the middle is comprised of a set of neurons connected by fixed synapses generated randomly [24]. As multiple recurrent loops are created by these synaptic connections, the reservoir features transient behavior that memorizes information of its inputs in the past. Reservoir neurons and readout neurons are connected by plastic synapses whose weights are to be adjusted according to the adopted learning

rule. Through its plastic synapses, each readout neuron receives a weighted sum of input signals from the reservoir neurons.

From Fig. 1, it is clear that the input signals are processed in two steps. The first step involves input neurons, reservoir neurons, and synapses connecting these neurons. Since the number of neurons in the reservoir is generally larger than that of the neurons providing inputs, in this step, the reservoir maps each input signal to its liquid response, a higher dimensional transient state. In the second step, the liquid response is projected to each readout neurons through plastic synapses

$$I_o(t) = \sum_i w_{i,o} \cdot r_i(t) \quad (1)$$

where t is time, $I_o(t)$ is the input to a readout neuron, $r_i(t)$ is the output of the i th reservoir neuron, and $w_{i,o}$ is the weight of the synapse connecting the i th reservoir neuron and the readout neuron. Over the duration of $[0, T]$ of an input temporal signal, the net integrated input to the readout neuron is

$$\int_0^T I_o(t) = \sum_i w_{i,o} \cdot \int_0^T r_i(t) \quad (2)$$

2.2 SNN Mapping Methods

Since the architecture of each neuromorphic platform is different, a dedicated toolchain is required to enable SNN to efficiently simulate on the neuromorphic platform. SpiNNaker [7] is a 2D toroidal mesh structure. PACMAN [8] was proposed to address SNN mapping on SpiNNaker. PACMAN uses a simulated annealing algorithm to search out the best partitioning scheme. But PACMAN only partitions the SNN model, which leads to spike congestion on the NoC. TrueNorth [1] also has their own mapping tool - *corlet* [2]. It uses the layout and routing optimization scheme in the traditional VLSI field for the mapping of logical SNNs to physical cores. SpiNemap [3] is proposed for the 2D-mesh architecture of Dynapse [23]. It divides the mapping process into two phases: partitioning and placement. They design a greedy Kernighan-Lin algorithm used in the partitioning phase and use the particle swarm optimization algorithm in the placement phase. For some neuromorphic platforms designed by new devices, [18, 25] were proposed to enable SNN to effectively run on these neuromorphic platforms.

3 LSM Mapping Method

3.1 Overview

We propose a LSM mapping method to efficiently deploy the LSM into the NoC-based neuromorphic platform. This work uses the SNN mapping toolchain - SNEAP that we proposed earlier [19]. The SNEAP only can process the feedforward SNNs, not LSMs with recurrent connections. The LSM mapping method

improves SNEAP, which is extended to support SNNs with recurrent connections. As shown in Fig. 2, LSM mapping method includes three steps: 1. *Profiling*: The SNN software simulator will randomly generate the topology of reservoir and then train the LSM. The topology of reservoir and spike traces are extracted to form an undirected traffic graph; 2. *Partitioning*: Using a multi-level partitioning algorithm divides the graph into multiple partitions. It can minimize the number of spikes between partitions within the limited capability and resource of target hardware; 3. *Mapping*: The SA algorithm was used to search for a mapping scheme to map these partitions to the NoC-based neuromorphic platform, which minimizes the average-hop of all spikes on NoC of target hardware.

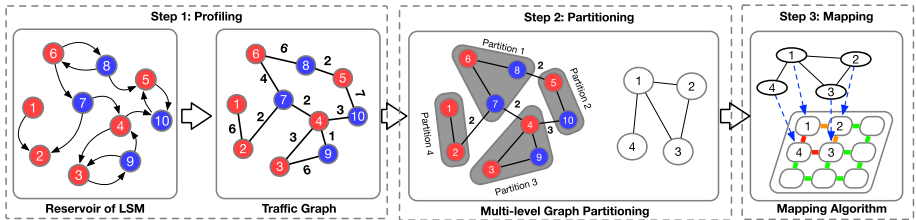


Fig. 2. Overview of LSM mapping method.

3.2 Profiling

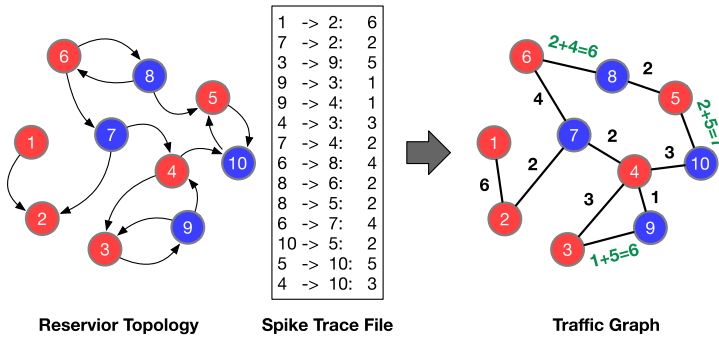


Fig. 3. Workflow of profiling step, spike trace file format: (Source neuron ID \rightarrow Destination neuron ID: The number of spikes).

In this step, we use the SNN software simulator (Carlsim [4], Brian2 [22] and etc.) to accurately simulate the behavior of the LSM. Most of SNN software simulators provide programming interfaces to configure LSM information. We use these interfaces to deploy the LSM, which provides it with some attribute information of the LSM, such as the number of neurons, neuron dynamic model, and network topology.

After the deployment, we extract the topology of the reservoir to form a graph, treating neurons as nodes and synapses as edges. Because there is a recurrent connection in the reservoir, the extracted graph is a directed graph, that is, bidirectional edges will occur between two neurons, but the connection between the neurons is unidirectional in the feedforward neural network. When the LSM simulation finished, the simulator will generate spike behavior files. These files contain all spike traces during the simulation. Each spike trace contains the ID of the source and destination neurons and firing time. The weight of each edge in the directed graph is the number of spikes communicated on the edge. Finally, the weight of the bidirectional edge in the directed graph is combined to generate an undirected traffic graph. The whole workflow is shown in Fig. 3.

SNEAP use Carlsim to construct the traffic graph of SNNs and spike trace file. In this paper, we use Brian2 to extract the topology of reservoir and spike behavior. When the simulation is finished, the traffic graph and spike trace files are generated by analyzing the files generated by Brian2. Then we use the traffic graph and spike trace files to partition and map the reservoir of LSM onto the NoC-based neuromorphic platform.

3.3 Partitioning

In this step, we use a multi-level graph partitioning strategy [13] to build a partitioning algorithm. This algorithm can quickly partition the traffic graph of the LSM and optimize the number of spikes between partitions.

The partitioning problem of LSM is a classic graph partitioning problem. Previous work used classic algorithms to solve this problem, such as the Kernighan-Lin (KL) [15] algorithm, particle swarm optimization (PSO) [14] algorithm, etc. However, the time required for these algorithms increases significantly as the scale of the graph increases. The quality of the solutions found by these algorithms is lower than our proposed partitioning algorithm.

Our proposed algorithm consists of three steps: *Coarsening*, *Initial partitioning*, *Uncoarsening*.

Coarsening consists of multi-level operations. The initial LSM traffic graph is folded and compressed level by level. The folding process is to randomly select a node in the current graph, and then combine the node corresponding to the edge with the maximum weight among overall valid adjacent edges into a large node. After multi-level folding operations, the original graph is folded into a coarse graph.

Initial partitioning divides the coarse graph obtained in the previous step into k partitions, and the sum of the weights of the nodes in each partition cannot exceed the capacity of the neuromorphic core. A node is randomly selected from the coarse graph to join the partition, and then selected the node corresponding to the edge with the maximum weight among overall valid adjacent edges of the previous selected node is added to the partition. The set of adjacent edges of this partition is updated. This process is repeated until the weight of the nodes in

the partition reaches the upper limit capacity of the neuromorphic core. Finally, the coarse graph is divided into k partitions.

Uncoarsening is similar to the coarsening step, and it is also divided into multi-level operations. But the uncoarsening step is to expand the nodes level by level. The partitioned coarse graph will expand up to the initial graph. In each level of the expansion process, it is necessary to adjust the nodes in each partition to optimize the communication between the partitions. After such uncoarsening level by level, the optimized k partitions are finally obtained.

3.4 Mapping

After the LSM is divided into multiple partitions, the position of these partitions on the NoC-based neuromorphic platform also affects the energy consumption and latency of the platform. We use the SA algorithm to quickly find out a mapping scheme with lower energy consumption and lower latency. SA can converge faster and effectively avoid stuck at local optima compare to other heuristic search algorithms, such as PSO, Tabu, and etc.

The objective function of the search can be latency or energy consumption. However, evaluating these metrics usually requires the use of hardware or a hardware simulator, which will cause a lot of time overhead and make the entire search process unacceptable. Lee [17] states that the average hop count can be used for measuring the latency and energy consumption of NoC. Compared with the above two metrics, the average hop count is easier to obtain. For the XY routing algorithm, the number of hops between cores on NoC is the Manhattan distance, so average hop can be directly calculated. This method can avoid using hardware or hardware simulator, thereby reducing the corresponding time overhead.

The architecture of NoC-based neuromorphic platform can be considered as a graph $A(C, I)$, where C is the set of neuromorphic cores and I is the set of connections among these cores for a given interconnect topology. Mapping M can be transformed into $M : P(V, E) \rightarrow A(C, I)$. Mapping M is represented by a matrix $m_{ij} \in \{0, 1\}^{|C| \times |V|}$, where m_{ij} is defined as:

$$m_{ij} = \begin{cases} 1 & \text{if partition } c_i \in C \text{ is mapped to core } v_j \in V \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The optimization objective of SA is to find out the mapping scheme with the minimum average hop count AH , i.e.

$$M_{min} = \operatorname{argmin}\{AH(M_i) | i \in 1, 2, \dots, N\} \quad (4)$$

Where N is the number of evaluated mapping schemes.

The average hop count AH can be written as:

$$AH = \frac{\sum_i \sum_{j \neq i} V_{ij} d(i, j)}{\sum_i \sum_{j \neq i} V_{ij}} \quad (5)$$

where $d(i, j)$ is distance between router i and router j and V_{ij} is the number of spikes communicated between router i and router j .

For XY routing algorithm, the Manhattan distance d_{MH} is used to compute $d(i, j)$, i.e.,

$$d_{MH}(i, j) = |i_x - j_x| + |i_y - j_y| \quad (6)$$

Algorithm 1. SA-based mapping optimization algorithm

```

1:  $T_{initial}$  //initiate temperature
2:  $T_{min}$  //minimum value of temperature
3:  $k$  //times of iteration cycle
4:  $\theta = 0.97$  //the cooling rate of the temperature
5: initial.mapping = RandomMappingScheme()
6: better.mapping = initial.mapping
7: parent.mapping = initial.mapping
8: parent.averagehop = AH(parent.mapping)
9: while  $T > T_{min}$  do
10:   for  $0 \leq i < k$  do
11:     child.mapping = Disturb(parent.mapping) //exchange or move the partitions
12:     child.averagehop = AH(child.mapping)
13:     if child.averagehop < parent.averagehop then
14:       parent.mapping = child.mapping
15:       if child.averagehop < better.averagehop then
16:         better.mapping = child.mapping
17:       end if
18:     else
19:        $p = \exp(\text{parent.averagehop} - \text{child.averagehop}/T)$ 
20:        $r = \text{random}(0,1)$ 
21:       if  $r < p$  then
22:         parent.mapping = child.mapping
23:       end if
24:     end if
25:   end for
26:    $T = \theta \times T_{initial}$ 
27: end while

```

4 Experiment Setup

4.1 Experiment Platform

We build an experiment platform to evaluate the performance of the proposed LSM mapping approach. The experimental platform was constructed following two simulators.

Two simulators are SNN software simulator – Brian2 [22] and hardware simulator – extended Booksim2 [12]. Brian2 is a software SNN simulator that can be

used to train and test SNN networks. The behavior of spike can be analyzed from the log file of Brian. Extended Booksim2 is a trace-driven and cycle-accurate NoC simulator. We extend it to support the NoC structure of the neuromorphic platform and to communicate with spikes rather than data packets. Extended Booksim2 is used to simulate the execution of SNN on real NoC-based hardware, so as to evaluate key performance statistics of NoC, such as average hop, latency, and power consumption.

Our experiment uses three size of 2D-mesh NoC (4×4 NoC, 8×8 NoC and 16×16 NoC), and neuromorphic core adopts crossbar structure.

4.2 LSM Application

In order to evaluate the effectiveness of LSM mapping method by using SNEAP, we use 3 realistic LSM applications, which are MNIST [16], NMNIST [9], and FSDD [10]. MNIST is grayscale handwritten number images dataset. NMNIST is the dynamic-vision-sensor (DVS) version of MNIST dataset. Free Spoken Digit Dataset (FSDD) is a free open speech dataset consisting of recordings of spoken digits in wav files at 8 kHz. We construct 3 size of LSM for every application, as shown in Table 1.

Table 1. LSM applications

LSM's name	The number of neurons	The number of spikes
NMNIST-400 [9]	400	4,738,428
NMNIST-800 [9]	800	17,814,687
NMNIST-1000 [9]	1,000	21,466,904
MNIST-400 [16]	400	2,059,330
MNIST-800 [16]	800	6,290,754
MNIST-1000 [16]	1,000	26,533,630
FSDD-400 [10]	400	5,083,309
FSDD-800 [10]	800	15,078,523
FSDD-1000 [10]	1,000	61,854,887

5 Results and Discussion

In this section, we compare the proposed method with some state-of-the-art methods proposed by SpiNeMap and SCO. We mainly conducted experiments on the scalability and performance of the proposed method. The proposed method is compared with some state-of-the-art methods proposed by SpiNeMap and SCO on a set of LSM applications. SpiNeMap uses SpiNeCluster to partition SNNs into clusters to minimize the total number of spikes among the clusters and SpiNePlacer to optimize the placement of clusters to crossbars of the

neuromorphic hardware to minimize energy consumption and latency. SCO uses its framework to balance the utilization of crossbars in the hardware. We now describe these results in detail.

5.1 Scalability

To evaluate the scalability of the proposed method, we performed several experiments with LSM sizes ranging from 400 to 1000 neurons. In Fig. 4, we compare the global traffic (the number of spikes among partitions) and the execution time of partitioning step under different methods normalized to original(do nothing with original LSM) when the LSM size scales up for different applications. To avoid the impact of NoC size, we use a 4×4 2D-mesh NoC in these experiments. For example, as shown in Fig. 4(a), when the LSM size scales up for MNIST, the execution time of the proposed method only increases slightly, but SpiNeMap increases exponentially. The cause is that the proposed method can quickly compress a graph of large size LSM to reduce the execution time of the partitioning step. The proposed method has 3% fewer average the number of spikes among partitions than SpiNeMap. This result is similar to the improvements obtained for other applications.

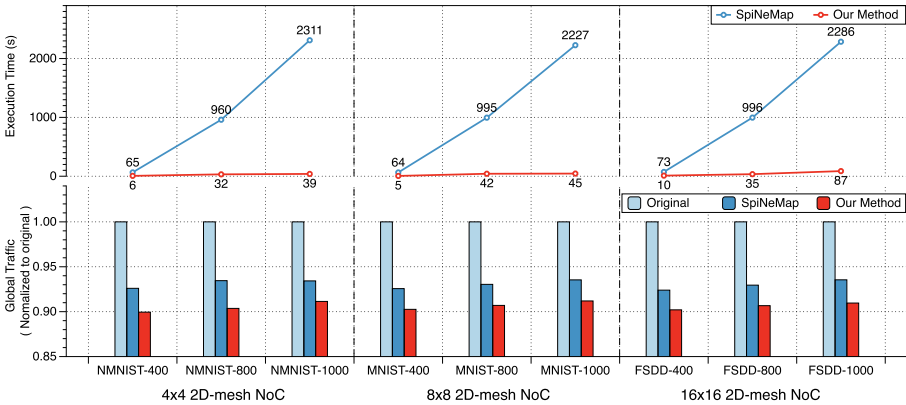


Fig. 4. Global traffic and execution time of partitioning step for increasing LSM sizes.

5.2 Performance

Latency. Figure 5 gives shows the latency on 2D-mesh NoC under different method normalized to SCO. The statistic shows that compared with SpiNeMap and SCO, the proposed method has a great reduction in all of LSM applications. The proposed method results in average 23% lower than the SpiNeMap and 74% lower than SCO on 4×4 2D-mesh NoC. These improvements are because of the optimization objective of our method. Our method adopts objective to minimize the total number of spikes among the partitions and average hop.

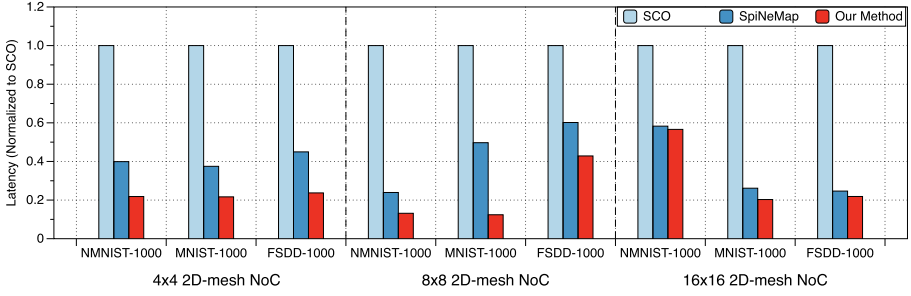


Fig. 5. Comparison of latency for increasing NoC sizes.

Energy. Figure 6 gives the energy of the NoCs under different methods normalized to SCO. Compared with other methods, Our method has the lowest energy consumption. For example, as shown in Fig. 6(b), the proposed results in average 57% lower than the SpiNeMap and 66% lower than SCO on 8×8 2D-mesh NoC. The improvement is due to the multi-level partitioning algorithm, which outperforms the greedy KL algorithm proposed by SpiNeMap. Fewer spikes communicated among the partitions, lower dynamic energy consumption.

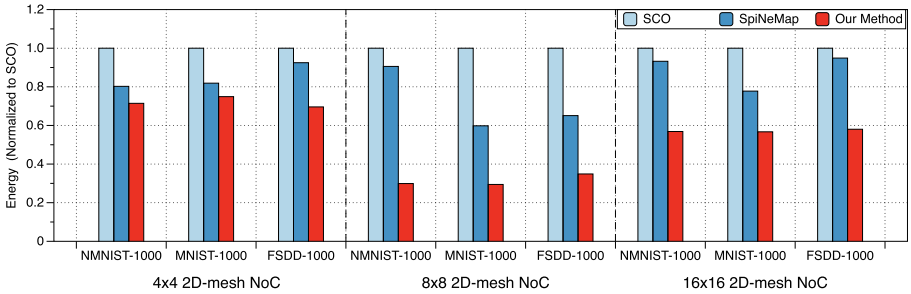


Fig. 6. Comparison of energy for increasing NoC sizes.

End-to-End Time. In Fig. 7, we illustrate the end-to-end execution time under different methods. Our method achieves $1.5\times$ lower average execution time than SpiNeMap. The causes behind this are that during the partitioning phase our method has a reduced amount of execution time compared to SpiNeMap and that in mapping phase SA converges faster than PSO.

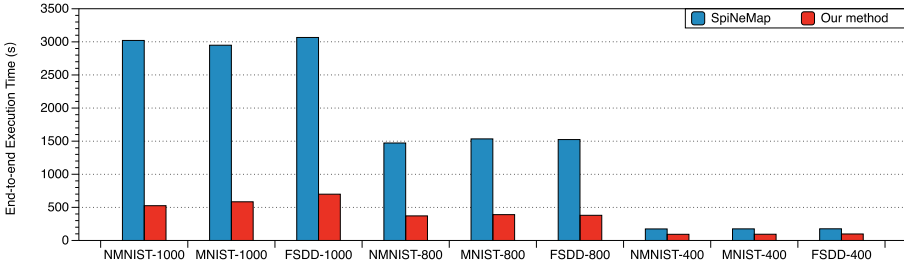


Fig. 7. End-to-end execution time.

6 Conclusion

This paper presents a fast and efficient LSM mapping method to map the large-scale LSM onto the NoC-based neuromorphic platform. The entire mapping method in three phases: Profiling, Partitioning, Mapping. Firstly, we use the SNN software simulator to extract the essential information of LSM such as topology and the behavior of spike. By using this information, we construct the undirected traffic graph and generate spike trace files. Then, the multi-level graph partitioning method is adopted to quickly divided the traffic graph into multiple LSM partitions. Our objective is to minimize the number of spikes between partitions. Finally, we use the SA algorithm to map optimized SNN partitions on the physical processing unit in hardware. Combining the optimization in the partitioning phase, mapping algorithm optimizes the energy consumption and spike latency on the NoC-based neuromorphic platform. Using different size of LSMs, we show that our method can achieve $1.5\times$ reduction in end-to-end execution time, and reduce average energy consumption by 57% on 8×8 2D-mesh NoC and average spike latency by 23% on 4×4 2D-mesh NoC, compared to SpiNeMap.

References

1. Akopyan, F., et al.: Truenorth: design and tool flow of a 65 mw 1 million neuron programmable neuromorphic chip. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **34**(10), 1537–1557 (2015)
2. Amir, A., et al.: Cognitive computing programming paradigm: a corelet language for composing networks of neuromorphic cores. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–10. IEEE (2013)
3. Balaji, A., et al.: Mapping spiking neural networks to neuromorphic hardware. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **28**(1), 76–86 (2019)
4. Chou, T.S., et al.: CARLsim 4: an open source library for large scale, biologically detailed spiking neural network simulation using heterogeneous clusters. In: *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE (2018)
5. Davies, M., et al.: Loihi: a neuromorphic manycore processor with on-chip learning. *IEEE Micro* **38**(1), 82–99 (2018)

6. Diehl, P.U., Zarella, G., Cassidy, A., Pedroni, B.U., Neftci, E.: Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware. In: 2016 IEEE International Conference on Rebooting Computing (ICRC), pp. 1–8. IEEE (2016)
7. Furber, S.B., et al.: Overview of the spinnaker system architecture. *IEEE Trans. Comput.* **62**(12), 2454–2467 (2012)
8. Galluppi, F., Davies, S., Rast, A., Sharp, T., Plana, L.A., Furber, S.: A hierarchical configuration system for a massively parallel neural hardware platform. In: Proceedings of the 9th Conference on Computing Frontiers, pp. 183–192. ACM (2012)
9. Garrick, O., Ajinkya, J., Cohen, G.K., Nitish, T.: Converting static image datasets to spiking neuromorphic datasets using saccades. *Front. Neurosci.* **9**, 437 (2015)
10. Jackson, Z.: Free spoken digit dataset. <https://github.com/Jakobovski/free-spoken-digit-dataset>. Accessed 4 Dec 2019
11. Ji, Y., et al.: Neutrams: neural network transformation and co-design under neuromorphic hardware constraints. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 21. IEEE Press (2016)
12. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 86–96. IEEE (2013)
13. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.* **48**(1), 96–129 (1998)
14. Kennedy, J.: Particle swarm optimization. In: Encyclopedia of Machine Learning, pp. 760–766 (2010)
15. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* **49**(2), 291–307 (1970)
16. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998)
17. Lee, H.G., Chang, N., Ogras, U.Y., Marculescu, R.: On-chip communication architecture exploration: a quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)* **12**(3), 23 (2007)
18. Lee, M.K.F., et al.: A system-level simulator for RRAM-based neuromorphic computing chips. *ACM Trans. Archit. Code Optim. (TACO)* **15**(4), 64 (2019)
19. Li, S., et al.: Sneap: a fast and efficient toolchain for mapping large-scale spiking neural network onto NOC-based neuromorphic platform. In: Proceedings of the 2020 on Great Lakes Symposium on VLSI (2020, to be published)
20. Maass, W.: Networks of spiking neurons: the third generation of neural network models. *Neural Netw.* **10**(9), 1659–1671 (1997)
21. Maass, W., Natschläger, T., Markram, H.: Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput.* **14**(11), 2531–2560 (2002)
22. Marcel, S., Romain, B., FM, G.D.: Brian 2, an intuitive and efficient neural simulator. *eLife* **8**, e47314 (2019)
23. Moradi, S., Qiao, N., Stefanini, F., Indiveri, G.: A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE Trans. Biomed. Circuits Syst.* **12**(1), 106–122 (2017)
24. Natschläger, T., Maass, W., Markram, H.: The “liquid computer”: a novel strategy for real-time computing on time series. *Special Issue Found. Inf. Process. TELEMATIK* **8**(ARTICLE), 39–43 (2002)
25. Xia, Q., Yang, J.J.: Memristive crossbar arrays for brain-inspired computing. *Nat. Mater.* **18**(4), 309–323 (2019)



Compiler Optimizing for Power Efficiency of On-Chip Memory

Wei Wu^(✉), Qi Zhu, Fei Wang, Rong-Fen Lin, and Feng-Bin Qi

Jiangnan Institute of Computing Technology, Wuxi 214083, China
ww7tc163@163.com

Abstract. As we all know, power constraint is the biggest challenge to build an exascale computing system. Among all parts of high performance processor, on-chip memory, including register, cache and so on, is accessed frequently and incurs high power consumption during program executing. Due to trivial overhead and good portability, compiling is a promising way to reduce power consumption and thermal dissipation of processor. In this paper, we focus on compiling to save power of on-chip memory access. A compiler optimizing on bypassing registers is proposed to reduce the number of register access in order to lower the power of the register files. Besides that, to save the power consumption of cache, another compiler optimizing is proposed to elegantly adjust loop transformation to make a better use of L0 cache. Finally, in order to evaluate the effectiveness of the above techniques, we build a systematic evaluation platform, named as GEAT, which consists of compiler, performance simulator and power simulator. Experiment results show that our proposed techniques can effectively reduce the power consumption of on-chip memory with trivial overhead of performance.

Keywords: Compiler optimizing · Low power · On-chip memory · Exascale computing system

1 Introduction

Many countries and organizations, including U.S., Japan, E.U. and China, have proposed their developing maps of Exascale Computing. Comparing with the current supercomputers, there are stricter limitations on power efficiency, performance, programmability, reliability for the future exascale computers. Among these challenges, as we all know, power efficiency is the biggest one. For example, DoE of U.S. proposed that the performance to power ratio of an exascale computer should be at least 50GFLOPS/W [1], while the best system in the current GREEN500 [2], A64FX Prototype, can only achieve 16.8GFLOPS/W.

People have to redesign almost every part of the supercomputer to meet the power constrain of Exascale Computing. By breaking down the current computer systems, we can find that processors not only provide high performance, they also consume the most power of the whole system. What's more, in the processors, the power consumption of on-chip memory access is one of the biggest parts, which in turn affects the power level of

the whole chip. On the other hand, due to the limited area and frequent access pattern of the on-chip memory, the power density of which is extreme high, which could incur Heat Stoke [3] phenomenon and make processor working in the abnormal status. Therefore, additional power control circuit, more expensive chip package are introduced in the hardware to prevent processors from being degraded by local high thermal dissipation.

Besides the hardware supports, software efforts are indispensable for improving power efficiency. In the modern computer system, compiler is the interface between hardware and software. For a mainstream compiler, execution time, compilation time and object code size are the three optimizing targets. The focus of a traditional compiler is mainly on how to make programs running faster. Nowadays, compilers should take power into account. There are some questions should be answered elegantly in the Exascale era. How to schedule instructions to execute programs with less power consumption? How to support the innovation hardware efforts to save more power?

Inspired by the above questions, we propose compiler optimizing to make programs more couple with the power-efficient hardware mechanisms in the on-chip memory system. On one hand, an innovative compiler optimizing on bypassing registers is introduced. After analyzing the lifecycle of the register data, the compiler decides which data could be forwarded to the certain stage of the pipeline without updating the register files. By avoiding the abundant access to registers, saving the power consumption of register files. On the other hand, an innovative compiler optimizing on L0 cache is proposed. With an elegant control on loop transformation, the frequently-executed instructions are scheduled to stay in the L0 cache. By reducing the miss rate of L0 cache and the number of instruction fetching, power is saved. Finally, to evaluate the proposed optimizations, we build a systematic evaluation platform, which is based on Gcc, Gem5 and Mcpat, named as *GEAT*. Gem5 is used to simulate the architecture with the support of bypassing registers and L0 Cache. Mcpat simulates and evaluates the power consumption. And we implement the proposed optimizing in the Gcc source code.

The main contributions of this work are listed as follows:

1. An innovative compiler optimizing on bypassing register is proposed. Due to the bypassing registers and instruction scheduling, avoiding the unnecessary register access to improve the power efficiency of register files.
2. An innovative compiler optimizing on L0 cache is proposed. A performance-power tradeoff factor is introduced to build a loop evaluation model. With elegant loop transformation, we minimize L0 cache miss, and improve power efficiency.
3. A systematic simulator, *GEAT*, is proposed to evaluate the optimizing. *GEAT* consists of compiler, performance simulator and power simulator. It provides a practical basement for the architecture people to evaluate their power efficiency work in the Exascale era.

The rest of the paper is organized as follows. Section 2 introduces the related works. Section 3 describes the compiler optimizing on bypassing register. Section 4 shows the details of the compiler optimizing on L0 cache. *GEAT* is illustrated in Sect. 5. Experimental results and analysis are proposed in Sect. 6. Section 7 concludes the paper.

2 Related Work

In the 90s, low-power compiling has become an important research field in the architecture community. Many inspiring works have been done.

Compiler optimizing is implemented by passes. The mainstream compilers mainly focus on performance. The hierarchy of optimizing (e.g. -O0/-O1/-O2/-O3/-Ofast) is also organized concerning with performance. However, the coming of the Exascale era has upgraded the importance of power in the supercomputer system. More and more researches discussed how to evaluate the compiler optimizing on the metric of power. Lima et al. proposed *COSPpp* [4] to evaluate compiler optimizing on the metric of performance, power, and object code size, and to enable passes based on the different optimizing target. A similar method was carried out by Hesham et al. [5] to reorganize passes with the tradeoff between performance and power.

Reducing or eliminating the power consumption of the redundant units/progresses by compiling attracts lots of people's attention. Some research groups focused on the Power-gating method under the supervision of compiler [6, 7, 14–19]. Based on profiling, compiler issues customized control instructions to enable or disable certain processor units to throttling power. Due to the dramatic overhead of communication, Kathy et al. [8] proposed a code transformation method to lower the system power by minimizing the communication of inter-chip and intra-chip.

How to improve the power efficiency based on the on-chip memory system is an interesting research field. On one hand, people have proposed a lot of methods to reduce program's demand on register files. José et al. [7] transferred the redundant register files from the active status into the low-power status to reduced power consumption. Similarly, Shieh W. Y. et al. [9] throttled the voltage supply of register files with low utilization. Sanghyun Park implemented the On Demand RF Read [13] in the Intel XScale simulator to reduce the access number of register files. In this paper, a compiler optimizing is proposed to eliminate the redundant access to register files. With the help of bypassing registers, some data is directly forwarded to the certain stage of pipeline without updating registers.

On the other hand, some efforts have been made on the power efficiency of cache. It has been shown that reducing the block number of every cache accessing can effectively lower the power consumption of cache system [20–22]. Fang et al. [20] exploited software semantics in cache design. By detecting the cache-miss in advance, they avoid abundant associative searches to reduce dynamic power consumption. Jones et al. proposed a method [21] to map the frequent executed instructions to some fixed cache blocks, thus reducing the cache access power. Additional memory hierarchy can reduce the access number of low level cache [23–27], thus reducing power. Kin et al. [23] proposed to reduce L1 cache power consumption by introducing *filter cache* above L1 cache. Powell et al. predicted cache access pattern to reduce cache access number [24]. Ghosh et al. proposed a *Way Guard* [25] method to save power. With the help of *Bloom Filter* structure [28], cache controller can tell whether the requested data in the certain cache block or not, thus reducing the power consumption of cache block probing. In this paper, we proposed a compiler optimizing to elegantly adjust loop transformation to reduce L0 instruction cache miss rate, further to save power.

3 Compiler Optimizing on Bypassing Registers

The group of register files is an important memory hierarchy which is the closest one to the computing units. Tons of works have been done on how to map data in the registers to improve program execution performance. However, the influence of register files on system power consumption and system stability should be paid more attention.

We proposed a compiler optimizing on bypassing registers. Several compiler passes are used to identify the data which could be bypassed and mapped to the bypassing registers. With the help of instruction scheduling, more data would be suitable for the bypassing optimizing, thus avoiding more abundant register updating. As shown in Fig. 1, the proposed optimizing consists of two stages, register renaming (Sect. 3.1) and instruction scheduling (Sect. 3.2).

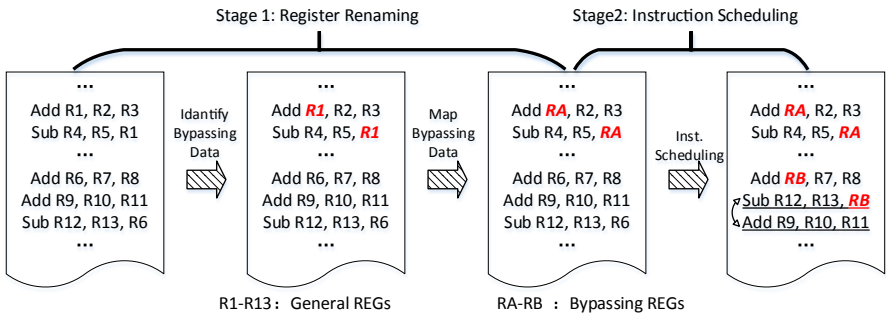


Fig. 1. Progress of compiler optimizing on bypassing registers.

3.1 Register Renaming Based on Bypassing

The purpose of register renaming is to remap data from general register to bypassing register, reducing the access number of registers files with the support of hardware bypassing mechanism.

The first step of renaming is to identify the data which could be mapped in the bypassing register. Lifecycle analysis and dependence analysis are performed at this step. Lifecycle analysis marks the registers in which data will not be used anymore. Dependence analysis can find out the instruction pair satisfying the data dependent constrains, which are listed as follows.

1. There is RAW (Read-After-Write) dependence between *INSN_A* and *INSN_B*. For example, the data in *REG_A* is updated by *INSN_A* firstly, and then is read by *INSN_B*.
2. The distance between *INSN_A* and *INSN_B* is less than a threshold *DIS_N*.
3. The data in *REG_A* will not be used anymore after the accessing of *INSN_A* and *INSN_B*.

In this paper, *REG_A* represents a general register, *INSN_A* and *INSN_B* represent instructions, and *DIS_N* is a configurable threshold describing the distance between two instructions.

First, lifecycle analysis is taken by a compiler pass. If the data in *REG_A* is not used anymore, assuming *REG_A* is in *INSN_A*, *REG_A* will be marked as *REG_DEAD* in the intermediate representation (*IR*) of *INSN_A*. It should be noticed that the interaction among compiler passes is complicated. The tag, *REG_DEAD*, may be marked insanely or be eliminated in the other passes. To ensure the correctness of lifecycle analysis, if the status of register is modified, the *REG_DEAD* tag should be updated accordingly.

Then, another compiler pass is introduced to probe the dependence between instructions related to the data in *REG_A*. It finds out the instruction pairs, e.g. *INSN_A* and *INSN_B*, which meet constraints listed above. A diagnosis is carried out to see whether the distance between *INSN_A* and *INSN_B* is less than *DIS_N*. If so, the tag of *REG_A* will be transferred from *REG_DEAD* to *REG_BYPASS*.

As shown in Fig. 1, *Add R1, R2, R3* and *Sub R4, R5, R1* are two instructions satisfying all the constraints. As a result of the lifecycle analysis, *R1*, marked in red in Fig. 1, will be tagged as *REG_BYPASS* in the *IR* of both instructions.

The second step of renaming is to place data in bypassing registers. Several registers are reserved as bypassing registers in advance, and they will not be assigned as general registers, such as *RA* and *RB* in Fig. 1. The register tagged as *REG_BYPASS* will be assigned a bypass register. For example, *R1* is renamed as *RA* in Fig. 1. The bypassing registers are assigned in turn. If the lifecycle of one bypassed data is overlapped with that of another one, renaming will be forbidden to ensure semantics correctness.

3.2 Instruction Scheduling Based on Bypassing

In some cases, because the distance between instructions is longer than *DIS_N*, data cannot be transferred to bypassing registers. In this section, we proposed an instruction scheduling scheme to enlarge the range that bypassing could work.

In the mainstream compiler, instructions scheduling is based on multi-queue and priority mechanisms. Instructions are maintained in different queues. In each queue, instructions are ranked according to their priorities. The priority is determined by multiple factors. Briefly speaking, if an instruction is ready to issue, it will be moved to the *READY* queue. And the priority indicates the instruction's position in the *READY* queue. If an instruction is issued, it will be moved from the *READY* queue to the *ISSUE* queue.

In the scheduling scheme, the priority of instruction in the *READY* queue is dynamically updated according to data dependence between issued instruction and ready instruction. As shown in Algorithm 1, if an instruction is issued, it will be moved to the *ISSUE* queue. A compiler pass is introduced to probe instructions in the *READY* queue. If there is *RAW* dependence between the current probed instruction and one of the last *N* issued instruction, the current probed instruction will be assigned the highest priority. If not, nothing will be done. In Algorithm 1, *N* presents the range of scheduling. The large *N* is, the more bypassing happens, thus the more power is saved.

Algorithm 1. Instruction scheduling based on bypassing.

READY_LIST: *READY* queue
ISSUE_LIST: *ISSUE* queue
N: Windows size of scheduling
issue_list_tail(x): The *x*th instruction moved to the *ISSUE* queue.
is_RAW(x, y): If there is *RAW* dependence between *x* and *y* (*x* updates data and *y* read data), it will return 1. Other wise, 0 will be returned.
update_priority(insn_list, x): Update the priority of instructions in *insn_list* queue, and the priority of *x* is set as the highest one.

```

1: for insn_ready in READY_LIST
2: for index in (1...N)
3: insn_issue = issue_list_tail(index)
4: if is_RAW(insn_issue, insn_ready)
5: update_priority(READY_LIST, insn_ready)
    
```

In Fig. 1, after swapping the last two instructions, data in *R6* can be bypassed. So, *R6* is renamed as *RB* to enable the bypassing mechanism.

4 Compiler Optimizing on L0 Cache

L0 cache is an additional instruction buffer between L1 cache and pipeline. It over-matches the traditional cache in four aspects. First, L0 cache stores the decoded instruction. Loading instructions from L0 cache is free of the decoding price. Second, due to the small memory space, the static power of L0 cache is much lower than that of the others. Third, because locality of instructions is much better than that of data, the miss rate of L0 cache could be very small. So the dynamic power of L0 cache is also lower than the other cache levels. Fourth, when L0 cache is accessed, there is no need to compare the tags and data in each block.

In this section, a compiler optimizing is described to improve utilization of L0 cache. Based on the L0 cache structure, a performance-power tradeoff factor is introduced to parameterize loop transformations, including loop unrolling and loop peeling. Several modes of loop transformations are performed. A loop evaluation model is built to find the best mode with a given performance-power tradeoff factor.

The optimizing progress is shown in Fig. 2. Firstly, the performance-power tradeoff factor is determined. The factor indicates the favor of users, performance first or power first. Then, loop unrolling and loop peeling are performed based on the loop pattern and

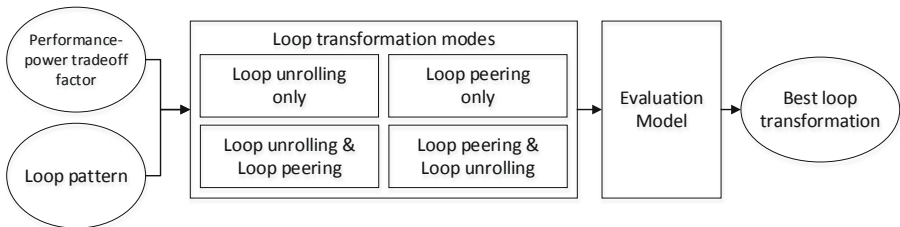


Fig. 2. Progress of compiler optimizing on L0 cache.

the factor. The loop evaluation model probes all loop transformation modes and finds the best one as output.

4.1 Performance-Power Tradeoff Factor

Traditional loop transformation mainly focuses on performance. Our proposed optimizing introduces the power metric, with a tradeoff between performance and power. Performance-power tradeoff factor, represented as *PERF_POWER_RATE*, is determined by users. It is delivered to compiler as a parameter, and is used to parameterize the loop transformations. The details of *PERF_POWER_RATE* are shown as follows.

1. The value of *PERF_POWER_RATE* is an integer which is no less than 0.
2. If *PERF_POWER_RATE* equals 0, the compiler optimizing on L0 cache will be disabled.
3. If *PERF_POWER_RATE* is larger than 0, the proposed optimizing will be enabled. The larger *PERF_POWER_RATE* is, the more performance is preferred in the optimizing. When *PERF_POWER_RATE* equals 1, power is the only metric considered in the optimizing.
4. *PERF_POWER_RATE* affects the number of loop unrolled, the block size of loop peeled, and how loop evaluation model works.

4.2 Loop Evaluation Model

Loop evaluation model is used to probe all the loop transformation modes and find the best one. The details of the model are described in the following statements.

For each loop transformation mode, the model first calculates the performance-power benefit, which can be expressed in formula (1).

$$LOOP_OPT_BEN = \frac{\sum_{i=1}^n ((INSN_i > \left(\frac{LOSize}{INSize} * PERF_POW_RATE\right)) ? 0 : 1)}{n} \quad (1)$$

In formula (1), n represents the number of loops which are transformed, $INSN_i$ means the number of instructions in the i^{th} loop, $LOSize$ indicates the size of L0 cache, and $INSize$ represents the memory size of each instruction.

Based on formula (1), the model finds out the maximum value of *LOOP_OPT_BEN*. And the according mode is the best one. The progress can be expressed in formula (2), in which m is the number of modes.

$$\max_{0 \leq x \leq m} LOOP_OPT_BEN_x \quad (2)$$

4.3 Loop Transformations Based on L0 Cache

In this subsection, we describe how loops are transformed based on L0 cache. There are two kinds of transformation, loop unrolling and loop peeling.

Instruction-level parallelism and L0 cache miss rate are both considered in the loop unrolling. Loop unrolling is a double sword. On one hand, loop unrolling frees instructions to be executed in the parallel way. On the other hand, too many instructions are unrolled from loops may exceeds the capacity of L0 cache, incurring high miss rate. Therefore, which loops are unrolled and how many loops are unrolled should be determined elaborately. Loop peeling transforms a big loop into several small loops. After that, there are more candidates could be placed in the limited space L0 cache with lower miss rate. The details of the proposed loop transformations are shown in Algorithm 2.

Algorithm 2. Loop transformations based on L0 cache

loop: The loop transformation mode to be analyzed.
loop_new: The loop transformation mode being analyzed.
best_loop: The best loop transformation mode.
LOOP_MODE_LIST: The queue of loop transformation modes.
loop_unroll_pass(x): Compiler pass performs loop unrolling.
loop_peel_pass(x): Compiler pass performs loop peeling.
compute_loop_ben(x): Evaluating the loop transformation mode.

```

1: for mode in LOOP_MODE_LIST
2:   loop_new = loop
3:   for sub_mode in mode
4:     if sub_mode == LOOP_UNROLL
5:       loop_new = loop_unroll_pass(loop_new)
6:     if sub_mode == LOOP_PEEEL
7:       loop_new = loop_peel_pass(loop_new)
8: LOOP_OPT_BEN = compute_loop_ben(loop_new)
9: if LOOP_OPT_BEN < MAX_OPT_BEN
10:  MAX_OPT_BEN = LOOP_OPT_BEN
11:  best_mode = mode
12:  best_loop = loop_new
13: loop = best_loop

```

Concretely, *loop_unroll_pass()* unrolls the inner most loop. The unroll time is determined by the loop unroll parameters (*MAX_INSNS*, *MAX_AVE_INSNS* and so on) and the performance-power tradeoff factor. Initially, the loop unroll parameters are set as the power preferred value. If *PERF_POW_RATE* is larger than 1, *MAX_INSNS* and *MAX_AVE_INSNS* will be multiplied by *PERF_POW_RATE* respectively. If *PERF_POW_RATE* equals 0, performance will be the only metric considered in the unrolling as the traditional way does. Based on the updated parameters, loop unrolling is performed.

In *loop_peel_pass()*, we first build a data dependence graph. The graph is probed to find out the producer-consumer case and the data independent case. Then, based on the size of L0 cache and *PERF_POW_RATE*, the best loop body size of peeling is determined. If *PERF_POWER_RATE* is no less than 1, the size of loop body will be set as L0 cache size/instruction size * *PERF_POW_RATE*. If *PERF_POW_RATE* equals 0, performance will be the only metric considered in the peeling as the traditional way does. Finally, based on the updated parameters, loop peeling is performed.

5 *GEAT*—A Systematic Simulator

In this section, we describe the systematic simulator *GEAT*, which is not only a platform to evaluate the above technologies, but also a practical basement for the architecture people to evaluate their power efficiency work in the future.

Figure 3 illustrates the framework of *GEAT*, which mainly consists of three parts, compiler, performance simulator and power simulator. Compiler is used to perform code transformation. Performance simulator consumes object code and produces performance data, which includes the basic execution parameters (in XML format). Power simulator takes the XML file as input, and reports the power consumption of each processor part.

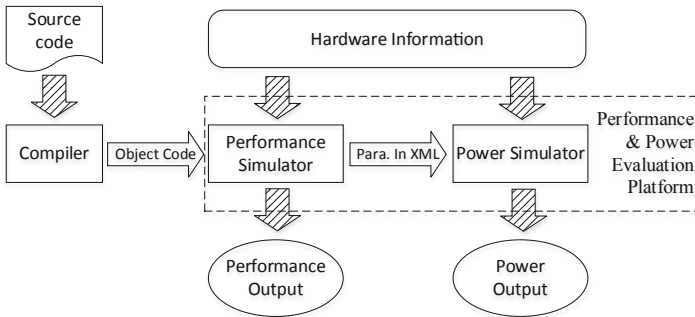


Fig. 3. Framework of *GEAT*.

Specifically, *GEAT* is implemented base on Gcc (compiler), Gem5 (performance simulator) and Mcpat (power simulator). In the source code of Gcc, several passes are inserted to implement the bypassing register optimizing and the L0 cache optimizing. Gem5 simulates a protocol processor, named as LPU (Low-Power CPU). The core unit of LPU is simple. So it is easy to scale the current LPU to a many-core architecture. Based on the parameters transferred from Gem5, Mcpat estimates the power consumed during program execution.

The performance & power evaluation platform is composed of the performance simulator and the power simulator. As shown in Fig. 4, the platform can be logically divided into three parts, which are performance model, interfaces and power model.

The performance model consists of multiple ISAs and a customized Gem5 simulator. By modifying the existing ALPHA ISA, we build LPU, a protocol ISA. Additional parts are inserted in the hardware structure to simulator bypassing registers and L0 cache. The number of bypass registers is set as 4. The capacity of L0 cache is configurable, and the default setting is 256B. In the power model, we also insert implements of bypassing registers and L0 cache in the IFU and EXU respectively, to estimate the power consumption of each LPU part. The interfaces can deliver the static architecture parameters, the dynamic execution information, including memory access time, execution cycles and system status.

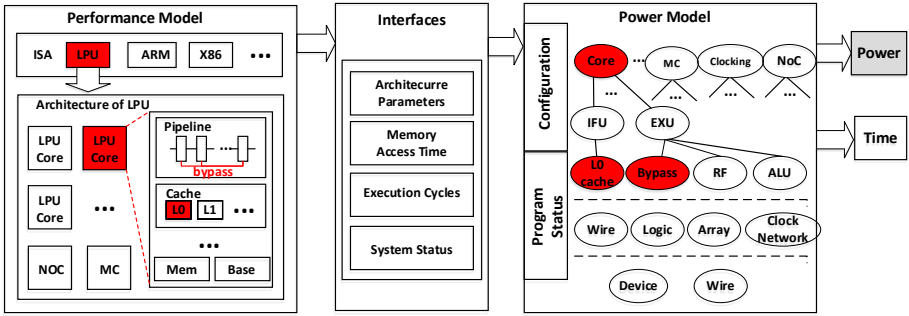


Fig. 4. Details of the performance & power evaluation platform.

6 Evaluation

In this section, we evaluate the proposed compiler optimizing on *GEAT*. The benchmark is NPB 3.1.1. And the size of Dataset is set as CLASS S. Both metrics of performance and power are evaluated.

6.1 Evaluation on Bypassing Register Optimizing

In this subsection, the power consumption of register files is evaluated. 4 bypassing registers are configured in the test. Figure 5 shows the power reduction of all ten NPB cases with the compiler optimizing on bypassing registers. The baseline is the power of the cases without optimizing. In Fig. 5, *A_n* represents the window size of scheduling. For example, *A₄* means the last 4 issued instructions are probed one by one, to see whether there are data dependence between them and the current instruction in the *READY* queue.

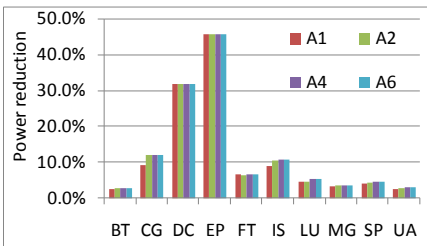


Fig. 5. Power reduction of register files.

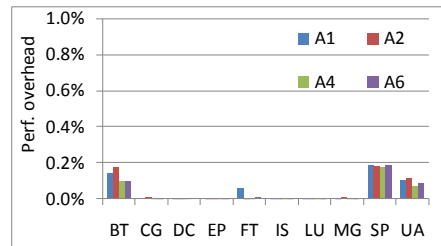


Fig. 6. Performance overhead of bypassing register optimizing.

As shown in Fig. 5, the power reduction of register files is remarkable. The largest reduction is 45%. There are 4 out of 10 cases save more than 10% power. And in the cases of *BT*, *CG*, *IS*, *LU*, *MG*, *SP* and *UA*, when the windows size of scheduling becomes bigger, more power can be saved. At most 3% power is reduced due to the more aggressive scheduling.

Figure 6 reveals the performance price paid for the power reduction. The baseline is the same as that in Fig. 5. And An represents the same as that in Fig. 5. It can be seen that in all of the ten cases, the overhead is trivial, which is no larger than 0.2%. The evaluation shows that the bypass register optimizing can effectively improve the power efficiency of register files.

6.2 Evaluation on L0 Cache Optimizing

First, we show the influence of L0 cache size on power efficiency. Both performance and power of CG are evaluated in Fig. 7. For power consumption, the value is normalized. The blue bars represent the case without optimizing, while the red bars represent the case with optimizing. $PERF_POW_RATE$ is set as 1. L0 cache size increases from 128B to 1024B. It is known to all that with a larger cache, the miss rate decreases, thus power consumption is reduced. Comparing the blue bars and the red bars, we can find that, with optimizing, 42% power is saved in the 128B setting, while 14.5% power is saved in the 1024B setting. Only about 3% power is saved due to the cache size increasing for the optimizing cases, which means that the proposed optimizing is not sensitive to the cache size. Smaller L0 cache prevents loops from being unrolled as described in Sect. 4, which incurs performance overhead. As the green line shows in Fig. 7, the largest overhead is 9% in the 128B setting. When L0 cache becomes larger, the overhead becomes trivial.

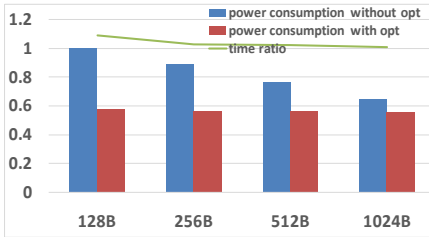


Fig. 7. The influence of L0 cache size.

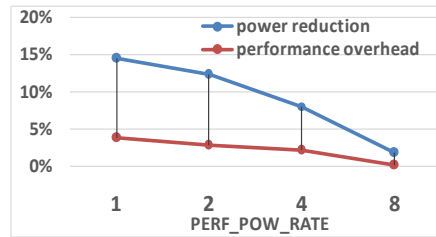


Fig. 8. The influence of $PERF_POW_RATE$.

It can be concluded that larger L0 cache brings better power efficiency. However, considering about the area overhead in the processor, a deliberate size of L0 cache should be the best choice.

Then, the effect of $PERF_POW_RATE$ is revealed, as shown in Fig. 8. We set the L0 cache size as 256B. The largest performance overhead case, UA , is taken as an example. And the baseline is when $PERF_POW_RATE$ equals 0, in which case, only the metric of performance is considered. It can be found that when $PERF_POW_RATE$ equals 1, the power reduction is about 15%, while the performance overhead is about only 4%. As $PERF_POW_RATE$ increases, less power is saved and the performance overhead becomes less too.

Finally, we provide an overall evaluation on L0 cache optimizing. In Fig. 9 and Fig. 10, L0 cache size is set as 256B and $PERF_POW_RATE$ equals 1. All ten cases of NPB are evaluated. The baseline is the cases without the proposed optimizing. Figure 9 reports the power reduction and Fig. 10 reveals the performance overhead. As shown

in Fig. 10, the largest power reduction is 36% in the CG case, and the average value is about 18%. In Fig. 10, it shows that the performance overhead is less than 3.9%, with an average of 1.8%. The evaluation proves that the optimizing is able to save power consumption with trivial performance overhead.

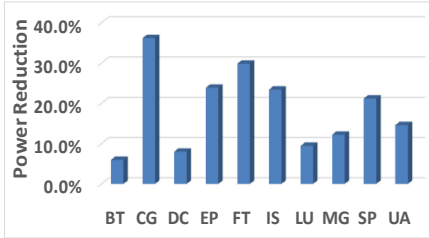


Fig. 9. Power reduction of instruction fetch unit.

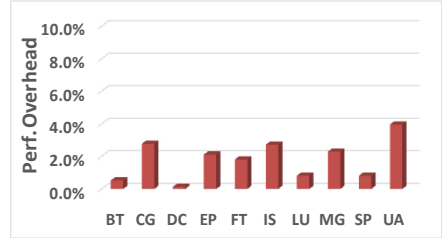


Fig. 10. Performance overhead of L0 cache optimizing.

7 Conclusion

In this paper, we focus on saving power consumption of the on-chip memory system by compiling. First, a compiler optimizing on bypassing registers is proposed to reduce the number of register accessing in order to lower the power of the register files. Besides that, to save the power consumption of cache system, we propose another compiler optimizing to elegantly adjust loop transformations to achieve better use of L0 cache. Finally, in order to evaluate the effectiveness of the above techniques, we build a power/performance estimating platform, named as *GEAT*, which consists of compiler, performance simulator and power simulator. Systematic evaluation reveals that our proposed techniques can effectively reduce the power consumption of the on-chip memory system with trivial performance overhead. More parts of processors will be investigated for power efficiency in our future work. Meanwhile, more aggressive compiler optimizing is developed undergoing.

Acknowledgments. This work is supported by the National Major Research and Development Program of China No. 2017YFB0202003.

References

1. Hemsoth, N.: Future challenges of large-scale computing. http://www.hpcwire.com/2013/04/15/future_challenges_of_large-scale_computing. Accessed 02 Mar 2020
2. The Green500 list – November 2019. <http://www.green500.org/greenlists>. Accessed 02 Mar 2020
3. Hasan, J., Jalote, A., Vijaykumar, T.N., Brodley, C.E.: Heat stroke: power-density-based denial of service in SMT. In: 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, pp. 166–177. IEEE (2005)

4. de Lima, E.D., de Souza Xavier, T.C., da Silva, A.F., Ruiz, L.B.: Compiling for performance and power efficiency. In: 2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Karlsruhe, Germany, pp. 142–149. IEEE (2013)
5. Hassan, H.H., Moussa, A.S., Farag, I.: Performance vs. power and energy consumption: impact of coding style and compiler. *Int. J. Adv. Comput. Sci. Appl.* **8**(12), 132–142 (2017)
6. You, Y.P., Huang, C.W., Lee, J.K.: Compilation for compact power-gating controls. *ACM Trans. Des. Autom. Electron. Syst.* **12**(4), 51 (2007)
7. Ayala, J.L., Veidenbaum, A., López-Vallejo, M.: Power-aware compilation for register file energy reduction. *Int. J. Parallel Prog.* **31**(6), 451–467 (2003)
8. Yelick, K.: Compiling to avoid communication. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 157–158. ACM, New York (2012)
9. Shieh, W.Y., Wang, B.S.: Power-aware register assignment for large register file design. *J. Supercomput.* **61**(3), 719–742 (2012)
10. Yun, H.S., Kim, J.: Power-aware modulo scheduling for high-performance VLIW processors. In: Proceedings of the 2001 International Symposium on Low Power Electronics and Design, Boston, USA, pp. 40–45. IEEE (2011)
11. Huff, R.A.: Lifetime-sensitive modulo scheduling. *ACM SIGPLAN Not.* **28**(6), 258–267 (1993)
12. Eichenberger, A.E., Davidson, E.S.: Stage scheduling: a technique to reduce the register requirements of a module schedule. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, USA, pp. 338–349. IEEE (1995)
13. Park, S., Shrivastava, A., Dutt, N., Nicolau, A., Paek, Y., Earlie, E.: Bypass aware instruction scheduling for register file power reduction. *ACM Sigplan Not.* **41**(7), 173–181 (2006)
14. Dropsho, S., Kursun, V., Albonesi, D.H., Dwarkadas, S., Friedman, E.G.: Managing static leakage energy in microprocessor functional units. In: 35th Annual IEEE/ACM International Symposium on Microarchitecture, Istanbul, Turkey, pp. 321–332. IEEE (2002)
15. Yang, H., Govindarajan, R., Gao, G.R., Cai, G., Hu, Z.: Exploiting schedule slacks for rate-optimal power-minimum software pipelining. In: Proceedings of the 3rd Workshop on Compilers and Operating Systems for Low Power, Charlottesville, USA, pp. 1–10 (2002)
16. You, Y.-P., Lee, C., Lee, J.K.: Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst.* **11**(1), 147–164 (2006)
17. You, Y.-P., Lee, C., Lee, J.K.: Compiler analysis and supports for leakage power reduction on microprocessors. In: Pugh, B., Tseng, C.-W. (eds.) LCPC 2002. LNCS, vol. 2481, pp. 45–60. Springer, Heidelberg (2005). https://doi.org/10.1007/11596110_4
18. Rele, S., Pande, S., Onder, S., Gupta, R.: Optimizing static power dissipation by functional units in superscalar processors. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 261–275. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_19
19. Zhang, W., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., De, V.: Compiler support for reducing leakage energy consumption. In: 2003 Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, pp. 1146–1147. IEEE (2003)
20. Fang, Z., et al.: Reducing L1 caches power by exploiting software semantics. In: Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design, Redondo Beach, USA, pp. 391–396. ACM (2012)
21. Jones, T.M., Bartolini, S., De Bus, B., Cavazos, J., O’Boyle, M.F.: Instruction cache energy saving through compiler way-placement. In: Proceedings of the Conference on Design, Automation and Test in Europe, Munich, Germany, pp. 1196–1201. ACM (2008)
22. Yu, C., Peter Petrov, P.: Aggressive snoop reduction for synchronized producer-consumer communication in energy-efficient embedded multi-processors. In: Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Co-design and System Synthesis, Salzburg, Austria, pp. 245–250. ACM (2007)

23. Kin, J., Gupta, M., Mangione-Smith, W.H.: The filter cache: an energy efficient memory structure. In: Proceedings of 30th Annual International Symposium on Microarchitecture, Research Triangle Park, USA, pp. 184–193. IEEE (1997)
24. Powell, M.D., Agarwal, A., Vijaykumar, T.N., Falsafi, B., Roy, K.: Reducing set-associative cache energy via way-prediction and selective direct-mapping. In: Proceedings 34th ACM/IEEE International Symposium on Microarchitecture, Austin, USA, pp. 54–65. IEEE (2001)
25. Ghosh, M., Ozer, E., Ford, S., Biles, S., Lee, H.H.S.: Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In: Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design, San Francisco, USA, pp. 165–170. ACM (2009)
26. Memik, G., Reinman, G., Mangione-Smith, W.H.: Just say no: benefits of early cache miss determination. In: The 9th International Symposium on High-Performance Computer Architecture, Anaheim, USA, pp. 307–316. IEEE (2003)
27. Zhang, M., Chang, X., Zhang, G.: Reducing cache energy consumption by tag encoding in embedded processors. In: Proceedings of the 2007 International Symposium on Low Power Electronics and Design, Portland, USA, pp. 367–370. ACM (2007)
28. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* **8**(3), 281–293 (2000)



Structural Patch Decomposition Fusion for Single Image Dehazing

Yin Gao¹, Hongyun Li², Yijing Su¹, and Jun Li¹(✉)

¹ Quanzhou Institute of Equipment Manufacturing, CAS, Quanzhou, China
yngaoyin@163.com, junli@fjirsm.ac.cn

² Quanzhou Institute of Technology, Quanzhou, China

Abstract. In this paper, we present a new image dehazing method via structural patch decomposition image fusion, which does not rely on the accuracies of global atmospheric light and transmission. Instead of estimating the exact global atmospheric and the transmission separately as most previous methods, our method directly constructs initial dehazing images with different exposure through the histogram analysis and structural patch decomposition image fusion filter to improve the visual dehazing effect. Experimental results show that this method outperforms state-of-the-art haze removal methods in terms of both efficiency and the dehazing visual effect.

Keywords: Image dehazing · Structural patch decomposition fusion · Adaptive boundary constraint · Multi-scale fusion

1 Introduction

In hazy weather, acquired images and videos suffer from loss of contrast and color cast and limit the visibility of targets in the scene [1]. This lacks in visibility can hinder the performance of computer vision-based systems such as outdoor surveillance, terrain analysis, and autonomous driving. Hence, image dehazing is an important topic and is being actively addressed to improve the safety of human and reduce traffic accidents by the research community.

Due to limited input information, single image dehazing is an ill-posed problem and more challenging [2]. Early single dehazing methods have two main types: enhancement-based method [3–6] and prior-based method [7–10] to solve this hazy problem. The former ones are mainly based on the human visual model [11, 12]. Due to the global convolution operation in this model, the dehazing image suffers from some color distortion. The latter mainly rely on the atmospheric scattering model. Due to huge computational complexity for the original model, He et al. [7] simplified it and firstly developed the dark channel prior (DCP) model. A large amount of literature about the DCP model has discussed image dehazing, which mainly focuses on solving the problems of color information, image contrast, the visible and algorithm complexity. These previous methods concentrated mainly on the use of various methods to optimize the transmission [13–16] and correct the DCP model [17–19].

Recently, learning-based methods are proposed to address the dehazing problem. Most of these works use deep networks to optimize the transmission and the atmospheric light directly [20–25]. Zhang et al. [23] proposed a pyramid densely connected transmission estimation network and a global atmospheric light estimation network respectively to dehaze. Engin et al. [24] presented a cycle-dehaze for image dehazing problem. More recently, Wang et al. [25] proposed an atmospheric illumination priori network through extensive statistical experiments for haze removal. These methods rely on accurate global atmospheric light and transmission. Moreover, big data sets are required to learn a large amount of parameter in the model, and the performance of these systems rely heavily on the quality of the dataset. However, lacking mated image pairs (haze image and the corresponding haze-free image), most existing methods use synthetic hazy images as training data which would result in undesired haze artifacts.

To solve these problems, image fusion is introduced to the field of image dehazing recently, which can improve the visual effects of the dehazing image effectively. Ancuti et al. [26] designed two input images by a white balance and a contrast-enhancing procedure and fused these two corresponding maps in a multi-scale fashion for image dehazing. This method is first introduced to the aspect of image dehazing but suffers from an unsatisfactory visual sense because of the insufficient fusion objects and the inaccurate global atmospheric light. Since then, many works have been published to dehaze. Recently, Gao et al. [27] presented a new image dehazing method by self-constructing image fusion, which avoids the interference of the global atmospheric light but increases the time-consuming.

To address these problems, we propose a structural patch decomposition image fusion strategy to dehaze. This work mainly addresses the challenging problem of visual effects in image dehazing. To obtain the effect range of the global atmospheric light, sky regions of the hazy image are segmented via the histogram analysis of hazy images. To optimize the transmission optimization, we propose a fast weighted least squares filter. Finally, to improve the visual effects of the dehazing image, these constructing initial dehazing images are blended via structural patch decomposition image fusion.

The main contributions of the proposed method can be summarized as follows. Firstly, we solve the effect range of the global atmospheric light instead of the exact value. Secondly, a fast weighted least squares filter with an adaptive boundary constraint is proposed to optimize the transmission for reducing the halo artifacts. Finally, a structural patch decomposition fusion method is proposed to improve the visual effects of the dehazing image.

2 Proposed Method

In this section, there are three major components during the dehazing. First, according to the histogram analysis of hazy images, we can obtain the effect range of the atmospheric light to construct several different exposure images. Subsequently, to properly optimize the transmission, the hazy image is processed by a fast weighted least squares filter with adaptive boundary constraint. Finally, we present a fusion method with structural patch decomposition to directly blend initial dehazing images with different exposure. Figure 1 shows the schematic diagram of the proposed dehazing method.

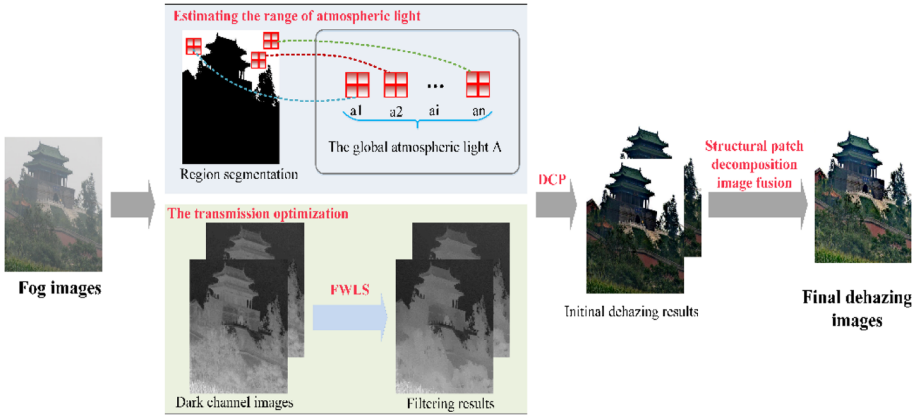


Fig. 1. Schematic diagram of the proposed dehazing method

2.1 Dark Channel Prior

In the field of image dehazing, the dark channel prior usually can be described as follows [7].

$$I(x) = J(x)t(x) + A(1 - t(x)), \tag{1}$$

where $I(x)$ is an observed or received image from a camera. $J(x)$ is an image without fog. $t(x)$ is the medium transmission describing the portion of the light that is not scattered and reaches the camera. A represents the global atmospheric light.

In the DCP, the transmission in (1) can be refined with the Guided image filtering.

$$t(x) = Gf(1 - \omega \cdot I_{min}(x)), \tag{2}$$

where $I_{min}(x) = \min_{c \in \{r, g, b\}} (\min_{y \in \Omega(x)} I_c(y) / A)$. $t(x)$ is the optimization result. $I_c(y)$ is the intensity of a channel of the RGB image. $Gf(\cdot)$ denotes the Guided image filtering method. ω is the constant parameter to keep a bit of haze for natural output appearance, $\omega \in (0, 1]$. The final scene radiance is represented by.

$$J(x) = (I(x) - A) / \max(t(x), t_0) + A, \tag{3}$$

where t_0 is the lower bound of the transmission $t(x)$.

2.2 Estimation of the Range of Atmospheric Light

To construct several different exposure images, we propose a new atmospheric light estimation method by performing histogram analysis on the observed fog image. We first perform smoothing processing with Gauss filtering on the histogram of each channel $I_c(x)$, and then obtain a segmentation threshold by the local minimum method.

$$\begin{cases} f_c(x) = G(I_c(x)) \\ a_c = \underset{x \in [0, 255]}{\operatorname{argmax}} (x | f'_c(x) = 0, f''_c(x) < 0), c \in \{r, g, b\}, \end{cases} \tag{4}$$

where $G(\cdot)$ is a Gaussian filter. a_c represents a threshold for segmenting sky regions in each channel of a fog image. $f_c(x)$ is the filtered result by $G(\cdot)$.

To estimate the range of the global atmospheric light A more effectively, we reformulated A by.

$$A = [\min(a'_c), \max(a'_c)], \quad (5)$$

where $a'_c = [\min(a_c), a_t, \max(a_c)]$. a_t is the range of the relatively high proportion values in the sky regions, which can be obtained via the histogram method. Figure 2 gives an example of the natural scene to show the sky regions' segmentation results. As illustrated in Fig. 2(a), (d) and (g) show the input hazy images. Figure 2(b), (e) and (h) show the segmentation results of these corresponding hazy images. Figure 2(c), (f) and (i) show our dehazing results.

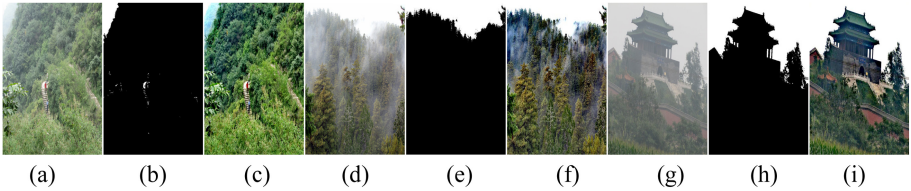


Fig. 2. The sky region segmentation results in different scenes. (a, d, g) The hazy image. (b, e, h) the corresponding segmented results. (c, f, i) our dehazing results

2.3 The Transmission Optimization

To optimize the transmission optimization, we propose a fast weighted least squares filter with an adaptive boundary constraint. According to the radiance cube definition, we define an adaptive boundary constraint of an arbitrary haze image with underneath translation.

$$t_i(x) = \min_{i \in [1, \dots, k]} \left\{ \max_{c \in [r, g, b]} \left(\frac{A_i - I_c(x)}{A_i - C_0^c(x)}, \frac{A_i - I_c(x)}{A_i - C_1^c(x)} \right) \right\}, \quad (6)$$

where $t_i(x)$ is the initial transmission with the boundary constraint in each global atmospheric light. $C_0^c(x)$ represents the minimum value of color channel pixel, $C_0^c(x) = \min_{c \in [r, g, b]} (I_c(x))$. $C_1^c(x)$ represents the maximum value of color channel pixel, $C_1^c(x) = \max_{c \in [r, g, b]} (I_c(x))$. A_i contains the range of the global atmospheric light by (5).

After the adaptive boundary constraint optimization, the transmission in (6) can be refined by a fast weighted least squares filter.

$$t'_i(x) = FW(t_i(x)), i \in [1, \dots, k], \quad (7)$$

where $FW(\cdot)$ denotes the function of the fast weighted least squares filter [28]. Figure 3 gives an example of transmission optimization. As illustrated in Fig. 3(a) shows the input hazy image. Figure 3(b) shows the corresponding initial transmission. Figure 3(c) shows corresponding optimized transmission. Figure 3(d) shows our dehazing results.

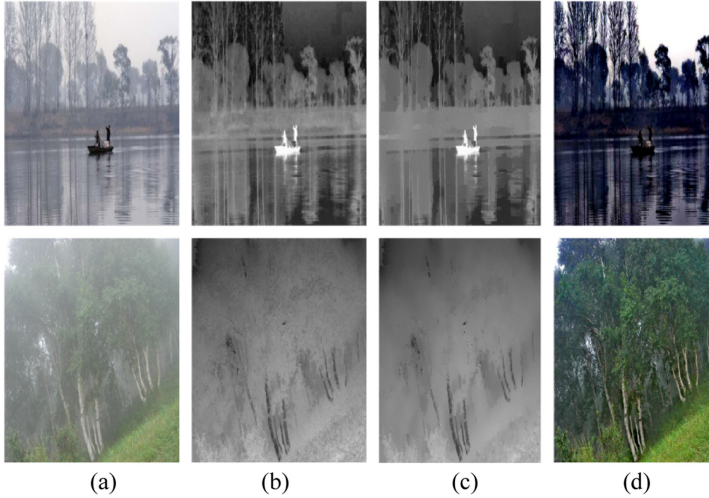


Fig. 3. The optimized results of the transmission. (a) The hazy image. (b) The initial transmission. (c) The optimized transmission. (d) Our dehazing results

2.4 Structural Patch Decomposition Image Fusion

To improve the better visual effects, the new multi-exposure fusion method is employed to improve the image quality. The fusion method typically follows a weighted summation framework [29].

$$J_f(x) = \sum_{i=1}^K J_i(x)W_i(x), \tag{8}$$

$$J_i(x) = (I(x) - A_i) / (\max(t'_i(x), \tilde{t}_0))^{dt} + A_i, i \in \{1, \dots, k\}, \tag{9}$$

where $J_f(x)$ is a dehazing image by our method. $W_i(x)$ represents a fusion weight map by structural patch decomposition. $J_i(x)$ is a dehazing result with different A_i and $t'_i(x)$.

For this fusion method, we will describe it in more detail in the following sections. This new fusion method is constructed by three components: local contrast, structural weight and color information.

$$J_f(x) = \hat{C} \cdot \hat{S} + \hat{L}, \tag{10}$$

where \hat{C} is the local contrast of all source image patches. \hat{S} represents the structures of all source image patches. \hat{L} is the color information of all source image patches.

Usually, the higher the contrast, the better the visibility. Considering that all input source image patches as realistic capturing of the scene, the patch that has the highest contrast among them would correspond to the best visibility. Therefore, the local contrast map can be calculated as follows.

$$\hat{C} = \max_{1 \leq i \leq k} C_i(x), \tag{11}$$

where $C_i(x) = \|J_i(x) - \mu_{J_i(x)}\|$, $\mu_{J_i(x)}$ is the mean value of the patch. $\|\cdot\|$ denotes the l_2 norm of a vector.

The desired structure of the fused image patch is expected to best represent the structures of all source image patches. A simple implementation of this relationship is given by.

$$\hat{S} = \frac{\bar{S}}{\|\bar{S}\|}, \quad \bar{S} = \frac{\sum_{i=1}^K f(v_i) \cdot s_i}{\sum_{i=1}^K f(v_i)} \quad (12)$$

where $f(v_i)$ denotes a weighting function that determines the contribution of each source image patch in the structure of the fused image patch, $f(v_i) = \|v_i^p\|$, $v_i = J_i(x) - \mu_{J_i(x)}$, p is an exponent parameter. $s_i = v_i / \|v_i\|$.

Concerning the color information of all source image patches, we take a similar function.

$$\hat{L} = \frac{\sum_{i=1}^K G(\mu_i, l_i) \cdot l_i}{\sum_{i=1}^K G(\mu_i, l_i)}, \quad (13)$$

where $G(\mu_i, l_i)$ denotes a weighting function, $G(\mu_i, l_i) = \exp\left(-\frac{(\mu_i - \mu_c)^2}{2\sigma_g^2} - \frac{(l_i - l_c)^2}{2\sigma_l^2}\right)$, $l_i = \mu_{J_i(x)}$, μ_i is the global mean value of the input image $J_i(x)$, σ_g and σ_l control the spreads of the profile along μ_i and l_i dimensions, respectively. μ_c and l_c are constants for the mid-intensity values.

3 Experimental Results and Analysis

To evaluate the effectiveness of our method, we compare our performance to state-of-the-art methods, such as He et al. [7], Berman et al. [10], Galdran [1], Bui et al. [9], Hu et al. [30], Shin et al. [31], respectively. All the methods are implemented on Windows PC with a Pentium Dual-core 2.4 GHZ CPU and 32.00 GB RAM using MATLAB2016a. To evaluate the dehazed performance quantitatively, we adopt the subjective and objective evaluation methods.

3.1 Qualitative Comparison of Natural Environment Images

For subjective evaluation, we test on several hazy visible images on the natural image dataset. Figure 4 shows the qualitative comparison with six state-of-the-art dehazing methods on color casts. Figure 4(a) shows the hazy images. Figure 4(b–g) depicts the results of six state-of-the-art dehazing methods, respectively. The results of the proposed method are given in Fig. 4(h).

As shown in Fig. 3, the results of He et al. [7] and Bui et al. [9] have the worst visual effect in the seven methods, The sky regions of these images significantly appear halo artifacts and suffer from over-enhancement in Fig. 4(b) and (e). The results of Berman et al. [10] have similar halo problems with the results of He et al. in Fig. 4(c), for instance, the sky regions of the dehazing images appear halo artifacts in fourth and sixth figures.



Fig. 4. Qualitative comparison of natural environment images. (a) The hazy images. (b) Ref. [7]. (c) Ref. [10]. (d) Ref. [1]. (e) Ref. [9]. (f) Ref. [30]. (g) Ref. [31]. (h) Our results.

The results of Galdran [1] have greatly improved the brightness of these images and increased the recognition of the target in these images, but the color fidelity is lost in Fig. 4(d). There is residual fog on the surface of these images in Fig. 4(d). The results of Shin et al. [31] are close to those observed by Galdran as displayed in Fig. 4(g). The color fidelity of those images has been greatly improved, but these results have remained a little haze in the image surface. The results of Hu et al. [30] achieve a better color fidelity, but still show slight over-enhancement in the target in these images. By the comparison of Fig. 4, our results have the best balance in color fidelity and visual effects. After our method processing, the objects of the six dehazed images can be recovered clearly, and the sky regions of these dehazed images are more natural.

3.2 Qualitative Comparison of Synthetic Hazy Images

In Fig. 5, seven methods including the proposed one in this paper are tested on synthetic images that are known for their rich colors and objects. Figure 5(a) shows the synthetic hazy images. The results of the six methods are shown in Fig. 5(b–g) respectively. The results of the proposed method are given in Fig. 5(h). Figure 5(i) is the ground truth image.

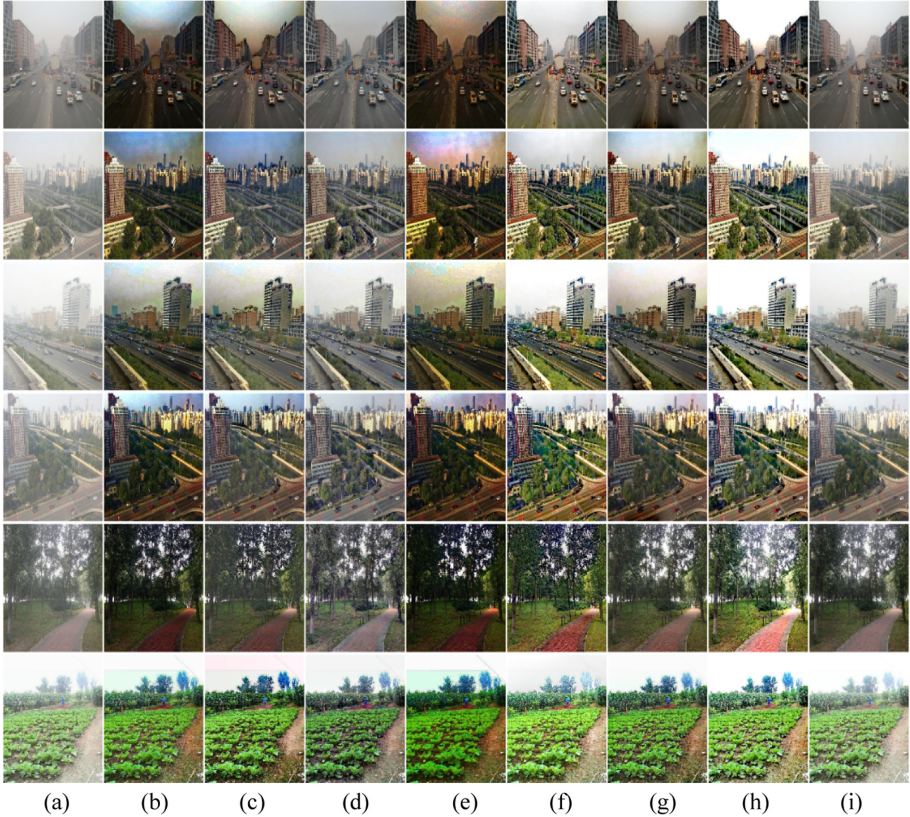


Fig. 5. Qualitative comparison of synthetic images. (a) The hazy images. (b) Ref. [7]. (c) Ref. [10]. (d) Ref. [1]. (e) Ref. [9]. (f) Ref. [30]. (g) Ref. [31]. (h) Our results. (i) Ground truth image

As shown in Fig. 5(b), the results of He et al. [7] remove most of the haze but significantly suffer from over-enhancement. The results of Bui et al. [9] have a similar problem as the results of He et al. [7] and Berman et al. [10] in Fig. 5(e) and Fig. 5(c), which tend to over enhance the local contrast of the image (see the sky regions in these images). The results of Shin et al. [31] improve the image brightness when dehazing, but it still appears fog edge and local over-enhancement in Fig. 5(g). As for the results of Galdran [1] and Hu et al. [30], they have a similar performance in the six images, which have been greatly improved in the visibility but have appeared local over-enhancement in the brightness regions. Compared with the ground truth image, the proposed method in this paper has the best visual recovering performance without halo artifacts and oversaturation.

3.3 Quantitative Comparison

To make a quantitative evaluation for the restoration performance, we make an experiment and evaluated these methods by two criteria: no-reference image quality assessment (ENIQA) [32] and Structure Similarity (SSIM) [33] in Fig. 5. The results are list in Table 1. Best and second-best results are marked in boldface.

Table 1. Quantitative results of the ENIQA and SSIM on real images.

	First	Second	Third	Fourth	Fifth	Sixth
N/S	0.107/–	0.005/–	–0.023/–	0.028/–	–0.066/–	0.030/–
Ref. [7]	0.089/0.663	0.096/0.757	–0.037/0.781	0.081/0.780	0.033/0.814	0.133/0.882
Ref. [10]	0.070/0.883	0.064/ 0.868	–0.084/0.851	0.061/0.895	0.036/ 0.905	0.097/0.815
Ref. [1]	0.024/ 0.941	–0.031/0.915	–0.079/ 0.958	0.097/0.922	0.007/0.870	0.080/0.876
Ref. [9]	0.120/0.632	0.097/0.694	0.008/0.701	0.118/0.697	0.073/0.705	0.079/0.751
Ref. [30]	0.050/0.837	0.058/0.775	–0.002/0.778	0.071/0.758	0.142/0.746	0.124/ 0.899
Ref. [31]	0.129/0.915	0.022/0.855	–0.100/0.837	0.069/0.906	0.203/0.964	0.111/0.874
Our	0.135/0.931	0.124/0.869	0.027/0.904	0.052/ 0.922	0.217/0.827	0.214/0.906

A higher ENIQA score indicates a more visual effect of the dehazed image. As can be seen from these results in Table 1, our results produce five higher ENIQA scores in the six images, followed by the results of Bui et al. which have three higher ENIQA scores. As for the results of He et al. and Galdran, they have consistent performance in the six images, which only produce one higher ENIQA scores. The results of Berman et al. are similar to the results of Hu et al., which do not produce one higher ENIQA score.

A higher SSIM score indicates that the dehazed image is closer to the ground truth image. As can be seen in Table 1, the results of our method have five higher SSIM scores in the six images, followed by Galdran which produces three higher scores. The results of Berman et al. produce two higher scores. The results of Hu et al. are similar to Shin et al., which produce one higher score. The results of He et al. are similar to Bui et al. which have the worst scores and do not produce higher scores.

4 Conclusion

In this work, we proposed a single image dehazing method via structural patch decomposition image fusion. The range of global atmospheric light can be estimated by the sky

region segmentation method with the histogram analysis of hazy images. Furthermore, a fast weighted least squares filter with an adaptive boundary constraint is constructed to optimize the transmission. Finally, a structural patch decomposition fusion method proposed to blend several dehazing images with different exposures. Experimental results show that our method performs favorably against some state-of-the-art methods on image visibility and color cast.

Acknowledgment. This work was supported by the Scientific and Technological Project of Quanzhou (No. 2019C009R, No. 2019C094R).

References

1. Galdran, A.: Image dehazing by artificial multiple-exposure image fusion. *Sig. Process.* **149**, 135–147 (2018)
2. Zhang, H., Sindagi, V., Patel, V.M.: Multi-scale single image dehazing using perceptual pyramid deep network. In: *IEEE Computer Social Conference Computer Vision Pattern Recognition Work*, Salt Lake City, USA, pp. 1015–1024. IEEE (2018)
3. Xiao, C., Gan, J.: Fast image dehazing using guided joint bilateral filter. *Vis. Comput.* **28**(6–8), 713–721 (2012)
4. Jobson, D.J., Rahman, Z.U., Woodell, G.A.: Properties and performance of a center/surround retinex. *IEEE Trans. Image Process.* **6**(3), 451–462 (1997)
5. Kapoor, R., Gupta, R., Son, L.H., Kumar, R., Jha, S.: Fog removal in images using improved dark channel prior and contrast limited adaptive histogram equalization. *Multi. Tools Appl.* **78**(16), 23281–23307 (2019). <https://doi.org/10.1007/s11042-019-7574-8>
6. Sun, Z., Han, B., Li, J., Zhang, J., Gao, X.: Weighted guided image filtering with steering kernel. *IEEE Trans. Image Process.* **29**, 500–508 (2020)
7. He, K., Sun, J., Tang, X.: Single image haze removal using dark channel prior. *IEEE Trans. Pattern Anal. Mach. Intell.* **33**(12), 2341–2353 (2011)
8. Dai, C., Lin, M., Wu, X., Zhang, D.: Single hazy image restoration using robust atmospheric scattering model. *Sig. Process.* **166**, 107257 (2020)
9. Bui, T.M., Kim, W.: Single image dehazing using color ellipsoid prior. *IEEE Trans. Image Process.* **27**(2), 999–1009 (2018)
10. Berman, D., Treibitz, T., Avidan, S.: Non-local image dehazing. In: *29th IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, USA, pp. 1674–1682. IEEE (2016)
11. Rahman, Z., Jobson, D.J., Woodell, G.A.: Multi-scale retinex for color image enhancement. In: *3rd IEEE International Conference Image Process*, Lausanne, Switzerland, pp. 1003–1006. IEEE (1996)
12. Jobson, D.J., Rahman, Z.U., Woodell, G.A.: A multiscale retinex for bridging the gap between color images and the human observation of scenes. *IEEE Trans. Image Process.* **6**(7), 965–976 (1997)
13. Lu, H., Li, Y., Serikawa, S.: Underwater image enhancement using guided trigonometric bilateral filter and fast automatic color correction. In: *2013 IEEE International Conference Image Processing*, Melbourne, Australia, pp. 3412–3416. IEEE (2013)
14. He, K., Sun, J., Tang, X.: Guided image filtering. *IEEE Trans. Pattern Anal. Mach. Intell.* **35**(6), 1397–1409 (2013)
15. Kim, J.H., Jang, W.D., Sim, J.Y., Kim, C.S.: Optimized contrast enhancement for real-time image and video dehazing. *J. Vis. Commun. Image Represent.* **24**(3), 410–425 (2013)

16. Meng, G., Wang, Y., Duan, J., Xiang, S., Pan, C.: Efficient image dehazing with boundary constraint and contextual regularization. In: 2013 IEEE International Conference on Computer Vision, Sydney, Australia, pp. 617–624. IEEE (2013)
17. Liao, Q., Yu, J.: Fast single image fog removal using edge-preserving smoothing. In: 2011 IEEE International Conference on Acoustics, Prague, Czech Republic, pp. 1245–1248. IEEE (2011)
18. Nishino, K., Kratz, L., Lombardi, S.: Bayesian defogging. *Int. J. Comput. Vis.* **98**(3), 263–278 (2012)
19. Chen, C., Do, M., Wang, J.: Robust image and video dehazing with visual artifact suppression via gradient residual minimization. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9906, pp. 576–591. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46475-6_36
20. Cai, B., Xu, X., Jia, K., Qing, C., Tao, D.: DehazeNet: an end-to-end system for single image haze removal. *IEEE Trans. Image Process.* **25**(11), 5187–5198 (2016)
21. Li, B., Peng, X., Wang, Z., Xu, J., Feng, D.: AOD-Net: all-in-one dehazing network. In: 2017 IEEE International Conference on Computer Vision, Venice, Italy, pp. 4770–4778. IEEE (2017)
22. Ren, W., Liu, S., Zhang, H., Pan, J., Cao, X., Yang, M.H.: Single image dehazing via multi-scale convolutional neural networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9906, pp. 154–169. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46475-6_10
23. Zhang, H., Patel, V.M.: Densely connected pyramid dehazing network. In: IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, USA, pp. 3194–3203. IEEE (2018)
24. Engin, D., Genc, A., Ekenel, H.K.: Cycle-dehaze: enhanced cycleGAN for single image dehazing. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, Salt Lake City, USA, pp. 825–833. IEEE (2018)
25. Wang, A., Wang, W., Liu, J., Gu, N.: AIPNet: image-to-image single image dehazing with atmospheric illumination prior. *IEEE Trans. Image Process.* **28**(1), 381–393 (2018)
26. Ancuti, C.O., Ancuti, C., Bekaert, P.: Effective single image dehazing by fusion. In: 2010 IEEE International Conference on Image Processing, Hong Kong, China, pp. 3541–3544. IEEE (2010)
27. Gao, Y., Su, Y., Li, Q., Li, H., Li, J.: Single image dehazing via self-constructing image fusion. *Sig. Process.* **167**, 107284 (2020)
28. Min, D., Choi, S., Lu, J., Ham, B., Sohn, K., Do, M.N.: Fast global image smoothing based on weighted least squares. *IEEE Trans. Image Process.* **23**(12), 5638–5653 (2014)
29. Ma, K., Li, H., Yong, H., Wang, Z., Meng, D., Zhang, L.: Robust multi-exposure image fusion: a structural patch decomposition approach. *IEEE Trans. Image Process.* **26**(5), 2519–2532 (2017)
30. Hu, H.M., Guo, Q., Zheng, J., Wang, H., Li, B.: Single image defogging based on illumination decomposition for visual maritime surveillance. *IEEE Trans. Image Process.* **28**(6), 2882–2897 (2019)
31. Shin, J., Kim, M., Paik, J., Member, S., Lee, S., Member, S.: Radiance-reflectance combined optimization and structure-guided L0-norm for single image dehazing. *IEEE Trans. Multimed.* **22**(1), 30–44 (2020)
32. Chen, X., Zhang, Q., Lin, M., Yang, G., He, G.: No-reference color image quality assessment: from entropy to perceptual quality. *arXiv Preprint arXiv:1812.10695*, pp. 1–12 (2018)
33. Wang, Z., Bovik, A.C., Sheikh, H.R., Simoncelli, E.P.: Image quality assessment: from error visibility to structural similarity. *IEEE Trans. Image Process.* **13**(4), 600–612 (2004)



Historic and Clustering Based QoS Aggregation for Composite Services

Zhang Lu and Ye Heng Zhou^(✉)

Guangxi Key Laboratory of Embedded Technology and Intelligent System,
Guilin University of Technology, Guilin, China
2103003882@qq.com

Abstract. Web services run in an open, heterogeneous and multi-tenant network environment, which makes the QoS of services uncertain and difficult to be described by a known probability distribution. Therefore, the calculation of QoS aggregation for composite services is facing challenges. This paper presents a new method for the aggregation calculation of composite services. In this method, the QoS of Web services is characterized by the sample space formed by their historical records, and a clustering method is adopted to control the number of samples in the sample space, so as to avoid the problem of combinatorial explosion during the process of aggregation calculation. This method does not need to limit the distribution of QoS, and is suitable for the composite services described by various common workflows and all kinds of QoS attributes. Experiments show that our method has advantages in terms of time cost and computational accuracy compared with the existing methods.

Keywords: Composite services · Uncertain QoS · Third keyword · Clustering

1 Introduction

Web services technology enables the formation of composite services by combining existing basic or complex services, so that applications can be deployed in distributed and heterogeneous environments. As there are more and more Web services on the network, quality of service (QoS) is used to describe the non-functional attributes of Web services, such as cost, response time, reliability or availability, which becomes an important selling point of Web services. QoS aggregation for composite services plays an important role in developing service-oriented applications.

Evaluation the QoS of a composite service accurately is of importance both to the service provider and consumer. Because Web services are in an open, heterogeneous, multi-tenant network environment, the QoS of Web services are probabilistic and not suitable to be described by a known statistical distribution. How to model and estimate the QoS of a composite service efficiently and accurately based on the QoS of its component services is still a critical challenge. In the following, service is used to refer to composite service. In this paper, the historical records of service QoS natures are used to describe

their QoS without limiting their distribution formation, and a clustering based QoS aggregation calculation method for composite services is proposed.

The remainder of this article is organized as follows: In Sect. 2, we discuss the work related to model and evaluation QoS of composite services. In Sect. 3, we provide the combined mode QoS operation and related symbols. In Sect. 4, a sample space clustering is designed. In Sect. 5, experiments are conducted and the correctness and performance of the proposed method are evaluated. Section 6 summarizes this article.

2 Related Work

When considering QoS aware Web service composition, many studies [1–4] ignore the uncertainty of QoS. However, the open, heterogeneous and multi-tenant net work environment makes the QoS of Web services uncertain, which gently increase the time complexity of QoS aggregation calculation. In literature [5], an example is given to illustrate that the response time of Web services does not follow any familiar probability distribution. Among them, PMF (Probability Mass Function) is used to describe the uncertainty of QoS. By requiring the sampling interval to have the same starting point and interval time, the impact of combinatorial explosion problem is reduced, and the time cost and calculation accuracy are improved. Literature [6, 7] respectively introduces two kinds of dynamic selection of Web services for QoS constraint decomposition. Skyline strategy [8, 9] is also used to serve service selection or service combination. It eliminates some candidate services or candidate combination schemes through probability analysis to narrow the exploration space faced by the later service combination algorithm. Such policies can only be considered as a pre-processing mechanism for service composition, and often need to limit workflow pattern (such as only supporting sequential pattern) [8] or limit distribution pattern [9]. Literature [10] proposes a global QoS service combination method based on decomposition. Literature [11] proposes to select the best available service by decomposing QoS constraints. Literature [6] regards QoS as a discrete random variable and uses Probability Mass Function (PMF) to describe it. Although PMF can describe any probability distribution, the author only uses a few field values to describe it, and the accuracy is low. Literature [12] proposes a dynamic selection of service composition based on time and QoS constraints. Literature [13, 14] assumes that QoS of Web services obeys normal distribution to simplify QoS aggregation calculation. Literature [15] proves that the use of specific statistical distribution to describe the uncertainty of service QoS is not rigorous enough. In literature [9], a few eigenvalues (expectation, entropy and hyperentropy) are adopted to represent the uncertainty of QoS. In essence, some eigenvalues of probability distribution are used to replace the probability distribution itself. Hwang [16] used the Probability Mass Function (PMF) to describe the QoS of atomic or composite services, and then calculated the aggregate QoS of composite services. This approach supports arbitrarily distributed QoS. In order to solve the combinatorial explosion problem in the calculation process, they refined an Aggregate Random Variable Discovery (ARVD) problem, and used dynamic programming and greedy algorithm to solve it. However, the accuracy of the scheme is low and the time cost is large.

3 Uncertain QoS Calculation for Composition Patterns

3.1 Underlying Assumptions

When calculating the QoS of composite services, the following assumptions are made:

- (1) the aggregate QoS of composite services depends on the QoS, composite mode and selection probability of the included services (for XOR mode).
- (2) the QoS of the services involved in the composite service is independent from each other.
- (3) QoS of the service is described by weighted samples with historical records.

3.2 QoS Aggregation Operations

A composite service can be described by a workflow pattern and typically involves four composite patterns: Sequential, Conditional, Parallel, Loop. QoS describing services can be divided into three categories: Additive (e.g., price, reputation), Multiplicative (e.g., reliability, availability), Max-operator (e.g., response time, service execution time). The three categories can convert most of the widely discussed QoS attributes. If $q(s, k)$ is used to represent the value of the k -th QoS attribute of service s , n represents the number of service included in a combined mode, and p_i represents the probability of choosing to perform service s_i when selecting the mode. Table 1 can be used to summarize the above four composite modes and QoS aggregation operations involved in three QoS types.

Table 1. QoS aggregation operation

QoS type	Additive	Multiplicative	Max-operator
Sequential	$\sum_{i=1}^n q(s_i, k)$	$\prod_{i=1}^n q(s_i, k)$	$\sum_{i=1}^n q(s_i, k)$
Conditional	$\sum_{i=1}^n p_i \cdot q(s_i, k)$	$\sum_{i=1}^n p_i \cdot q(s_i, k)$	$\sum_{i=1}^n p_i \cdot q(s_i, k)$
Parallel	$\sum_{i=1}^n q(s_i, k)$	$\prod_{i=1}^n q(s_i, k)$	$\max\{q(s_i, k)\}$
Loop	$n \cdot q(s, k)$	$q(s, k)^n$	$n \cdot q(s, k)$

The operations involved in Table 1 are summation, quadrature, weighted average, n -th power, and maximum value, all of which are in accordance with the exchange law and the combination law, so only need to consider the case when $n = 2$. Let $q(s_1, k)$ be represented by the sample space $X = \{u_i * x_i\} (i = 1, 2, \dots, r)$, where x_i is the i -th historical record and u_i is the weight given to x_i . $q(s_2, k)$ can be represented by the sample $Y = \{v_j * y_j\} (j = 1, 2, \dots, t)$, where y_j is the j th historical record and v_j is the weight given to y_j . If the sample space $Z = \{z_l\}$ is used to represent the aggregated QoS of the attribute k of s_1 and s_2 , according to Table 1, there are the following possibilities: $Z = X + Y, Z = X * Y, Z = p_1 * X + p_2 * Y, Z = \max\{X, Y\}$.

When $Z = X + Y$, z_l can be determined by formula (1). Where $i = 1, 2, \dots, r, j = 1, 2, \dots, t$:

$$Z_{l=i,j} = u_i \cdot v_j \cdot (u_i + v_j) \quad (1)$$

When $Z = X * Y$, z_l can be determined by formula (2)

$$Z_{l=i,j} = u_i \cdot v_j \cdot u_i \cdot v_j \quad (2)$$

When $Z = p_1 * X + p_2 * Y$, z_l can be determined by formula (3)

$$Z_{l=i,j} = u_i \cdot v_j \cdot (p_1 \cdot u_i + p_2 \cdot v_j) \quad (3)$$

When $Z = \max\{X, Y\}$, z_l can be determined by formula (4)

$$Z_{l=i,j} = u_i \cdot v_j \cdot \max\{u_i, v_j\} \quad (4)$$

The aggregate QoS of the overall composite service can be obtained by gradually aggregating the various combined modes from inside to outside.

4 Sample Space Clustering

As can be seen from Sect. 3.2, for a certain QoS, if the sample space describing QoS of a service contains m samples, the sample of QoS after the aggregation of two services will contain m^2 samples. An aggregate QoS sample space of a combined service containing n services will contain m^n samples, and there is a combination explosion problem. For this reason, when the number of samples contained in a sample space is too large, the clustering method needs to be used to limit the number of samples to the agreed range, so as to avoid the combination explosion problem in the aggregation calculation process.

Considering that the sample space for describing QoS is composed of unary data, Algorithm 1 is designed to cluster the sample space according to the k-Means [17] algorithm.

Algorithm 1 Clustering sample space.

Input: sample $X = \{ u_i * x_i \}, i = 1, 2, \dots, m$, the number of cluster $k < m$;

Output: sample $Y = \{ v_j * y_j \}, j = 1, 2, \dots, k$;

S1 set the initial clustering center $cs[j], j = 1, 2, \dots, k$;

S2 calculate the clustering center $p[j]$ to which x_i belongs, $i = 1, 2, \dots, m$;

S3 do {

S4 updates $cs[j]$ according to $p[j]$;

S5 calculate the moving distance D_{max} of the cluster center;

S6 update $p[j]$; }

S7 while ($D_{max} > \text{threshold}$);

S8 calculates v_j and y_j according to $p[j], j = 1, 2, \dots, k$;

S9 return $Y = \{ v_j * y_j \}, j = 1, 2, \dots, k$;

In S1, the initial clustering center cs is determined by formula (5), where k is the number of clustering centers, and max and min are the maximum and minimum values of the data set, and ε is randomly selected on $[-0.2, 0.2]$, so that cs has a certain randomness.

$$cs[i] = (i + 0.5 + \varepsilon) * \frac{max - min}{k}, i \in [0, k) \tag{5}$$

In S2, the clustering center to which x_i belongs can be calculated according to formula (6), $i = 1, 2, \dots, m$

$$x_i \in \begin{cases} p[0], x_i < ics[0] \\ p[k - 1], x_i > ics[k - 1] \\ p[j], ics[j] \leq x_i \leq ics[j + 1] \&\& 2x_i \leq ics[j] + ics[j + 1] \\ p[j + 1], ics[j] \leq x_i \leq ics[j + 1] \&\& 2x_i > ics[j] + ics[j + 1] \end{cases} \tag{6}$$

In S4, $cs[j]$ is updated to the average of all elements in $p[j]$. In S5, D_{max} is the maximum moving distance of the new and old cluster centers, and the threshold is set to $(max - min) / (10 * k)$.

In S6, if x_i originally belongs to $p[j]$, after the update, x_i may only belong to $p[j - 1]$, $p[j]$ or $p[j + 1]$, and the specific update rule is determined by formula (7).

$$x_i \in \begin{cases} p[j - 1], x_i < cs[j] \&\& 2x_i < cs[j - 1] + cs[j] \\ p[j + 1], x_i > cs[j] \&\& 2x_i < cs[j] + cs[j + 1] \\ p[j], \text{others} \end{cases} \tag{7}$$

In S8, y_j is the weighted average of the elements in $p[j]$, and v_j is the average of the weights of the elements in $p[j]$.

5 Experimental Analysis

In two scenarios, by comparing the method of [5] (marked as PDF) and the method of this article (marked as Cluster), analysis and verification of the effectiveness of the method in this article. In the Cluster method, the number of cluster centers is 50.

5.1 Scenario 1

This scenario considers the aggregate value of the response time of a combined service composed of n services combined in a sequential mode. Let the response time of each service be normally distributed $N(\mu, \sigma^2)$, μ takes random values in the interval [5, 10), and σ takes random values in [1, 2). Then the response time of the combined service is also normally distributed. Its exception is the sum of the expectations of each service, and its variance is the sum of the variances of each service. Based on the theoretical calculation of the expected and mean squared deviation of the aggregated QoS of the combined service, the relative difference between the expected and mean squared deviation obtained when using PDF (the sampling starting point is 0 and the sampling interval is 2) and this Cluster method can be calculated separately. Table 2 lists the relative difference between the expected and mean squared deviation of the combined service obtained by the two methods in 5 tests when n is equal to 100. It can be seen that the expected accuracy obtained by the Cluster method is nearly 20 times higher than that of the PDF, and the accuracy of the mean square error is about half of the PDF; the expected accuracy obtained by the Cluster method fluctuates slightly, and the accuracy of the mean squared error obtained by the Cluster method, and the expected and mean squared error obtained by the PDF are relatively stable.

Table 2. The relative difference of expectations and mean square errors

Seq	Expectation (%)		Mean variance (%)	
	Cluster	PDF	Cluster	PDF
1	-0.635	-14.267	14.131	6.245
2	-0.078	-12.768	15.309	6.865
3	0.043	-13.213	13.515	6.281
4	-0.234	-12.865	14.256	5.489
5	-0.096	-12.908	18.127	5.526

Table 3 compares the relative difference between the expected and mean squared deviations of the combined services obtained by the above two methods when n increases

from 50 to 350 in steps of 50 at a certain measurement. It can be seen from: the accuracy of the expected and mean square deviation obtained by the PDF method is basically not affected by the number of tasks; the expected accuracy obtained by the cluster method decreases with the increase of the number of tasks, which is much better than the PDF method; the accuracy of the mean square error obtained by the cluster method increases with the number of services. When the number of services reaches more than 200, it is close to the PDF method, and when the number of services reaches more than 300, it is much better than the PDF method.

Table 3. The accuracy varies with the number of services

Num	Expectation (%)		Mean variance (%)	
	Cluster	PDF	Cluster	PDF
50	-0.135	-12.633	8.274	4.063
100	0.242	-13.069	9.868	5.780
150	-0.511	-13.744	9.495	6.405
200	-0.644	-13.094	5.117	5.995
250	-1.150	-13.342	5.141	6.510
300	-1.015	-13.635	0.723	6.294
350	-0.968	-13.465	-0.844	5.889
200	-0.644	-13.094	5.117	5.995
250	-1.150	-13.342	5.141	6.510
300	-1.015	-13.635	0.723	6.294
350	-0.968	-13.465	-0.844	5.889

5.2 Scenario 2

The service composition problem with response time constraints is considered. The workflow is generated randomly according to the number of services it contains, involving three modes of order, concurrency and selection, with a ratio of about 2:1:1.

Ws-dream dataset [15] was used as the data source. It involves 4500 Web services, each of which involves two QoS attributes, response time and throughput, and each QoS of each service involves 64 real measurements from 142 users, i.e., each service contains about 9000 records. From this data set, 100 records were randomly selected for the response time of each Web service as QoS samples describing the response time of the service, and the sample weights were all equal. In the experiment, QoS samples of the candidate services are selected from these 4500 services in a sequential loop.

When the number of samples is high (2000 times the number of services), the results should be more reliable. With reference to the expectation and mean square error of

aggregate QoS of composite services obtained by sampling method (marked as Sample), the relative difference between expectation and mean square error obtained by PDF (sampling starting point is 0, sampling interval is 200) and Cluster method can be calculated. Since the workflow and candidate services are random, the results of each test will fluctuate, averaging the absolute value of the relative difference from 10 tests.

Figure 1 compares the expected relative difference between the two approaches with the number of services. It can be observed from Fig. 1 that Cluster and PDF methods are not significantly correlated with the number of services in accuracy. The relative difference of the method is mostly between 1–2%, which is much better than the PDF method (between 6–8%).

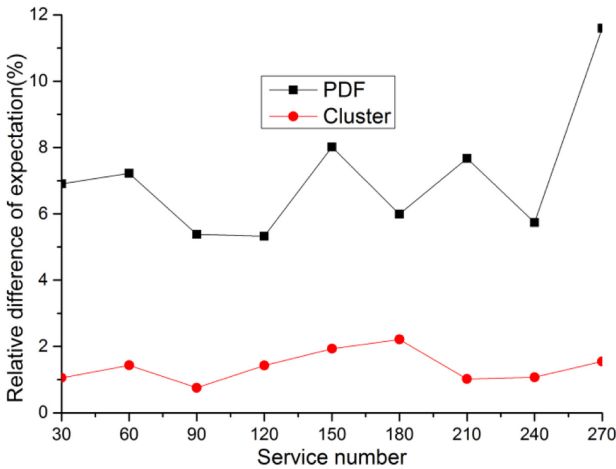


Fig. 1. The relative difference of expectations varies with the number of services

Figure 2 compares the variation of the mean square error relative difference obtained by the two methods with the number of services. It can be seen from Fig. 2 that, for the relative difference of mean square error, the Cluster method has little correlation with the number of tasks in the precision, while the PDF method has a decreased precision when the number of services is small and large. This is because the sampling interval distance of PDF method is an important parameter, which needs to be dynamically adjusted according to the number of tasks and other factors, while this experiment takes a fixed value. Compared with PDF, the relative difference of mean variance obtained by Cluster method also has certain advantages (the relative difference of Cluster method is between 1–4%, while PDF is more than 4%).

Figure 3 shows how the time cost of the three methods varies with the number of services. It can be seen that: the Sample method costs a lot of time; When the number of services is small (less than 40), The time cost of Cluster is close to that of PDF, but when the number of services is large, the time cost of Cluster is much lower than that of PDF. This is because the time cost of Cluster method increases linearly with the increase of the number of services, while the time complexity of PDF is roughly equal to the square of the number of services.

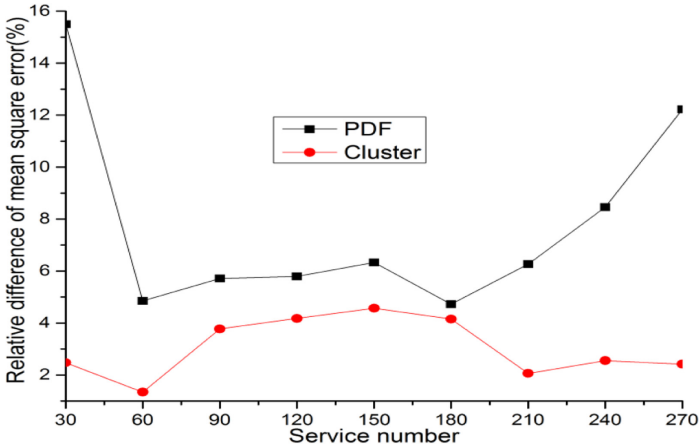


Fig. 2. The relative difference of mean square errors varies with the number of services

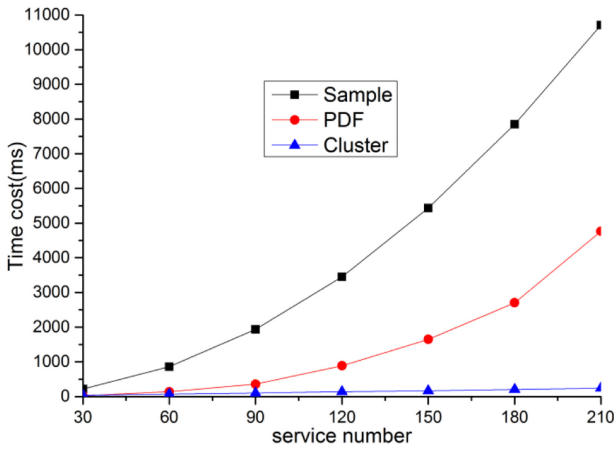


Fig. 3. Time cost varies with the number of services

Experimental analysis shows that compared with PDF method, Cluster method has obvious advantages in time cost and computing accuracy.

6 Conclusion

QoS aggregation with uncertainty plays an important role in service composition optimization. In this paper, by representing QoS uncertainty according to historical records and adopting a clustering algorithm similar to k-means to avoid the combinatorial explosion problem in the process of aggregation calculation, a new QoS aggregation method is proposed. This method only involves a few parameters such as the number of historical samples and the number of clustering, and can be used to any QoS attributes and common workflows. Simulation results show that our method is superior to existing methods.

Acknowledgment. This research work was supported by Guangxi University key Laboratory Director Fund of Embedded Technology and Intelligent Information Processing (Grand No. 2018A-05) and Foundation of Guilin University of Technology (Grand No. GUTQDJJ2002018).

References

1. Sellami, W., Hadj Kacem, H., Hadj Kacem, A.: Dynamic provisioning of service composition in a multi-tenant SaaS environment. *J. Netw. Syst. Manag.* **28**(2), 367–397 (2020). <https://doi.org/10.1007/s10922-019-09510-2>
2. Ehsan, A., Mahsa, M., Omid, F.V.: A novel model for optimisation of logistics and manufacturing operation service composition in Cloud manufacturing system focusing on cloud-entropy. *Int. J. Prod. Res.* **58**(7), 1987–2015 (2020)
3. Yan, H., Fu, X.D., Yue, K., Liu, L., Liu, L.J.: Uncertain QoS aware web service selection method using prospect theory. *J. Chin. Mini-Micro Comput. Syst.* **40**(05), 953–958 (2019)
4. Shen, J.Q., Luo, C.W., Hou, Z.W., Liu, Z.Z.: QoS aware logistics web service composition base on improved genetic algorithm. *J. Chin. Mini-Micro Comput. Syst.* **40**(01), 36–39 (2019)
5. Zheng, H., Yang, J., Zhao, W.: Probabilistic QoS aggregations for service composition. *ACM Trans. Web* **10**(2), 1–36 (2016)
6. Hwang, S.Y., Hsu, C.C., Lee, C.H.: Service selection for web services with probabilistic QoS. *IEEE Trans. Serv. Comput.* **8**(3), 467–480 (2015)
7. Wang, S.G., Sun, Q.B., Yang, F.C.: Web service dynamic selection by the decomposition of global constraints. *J. Softw.* **22**(7), 1426–1439 (2011)
8. Wang, X.S., Fu, X.D., Liu, L., Yue, K., Liu, L.J.: Probabilistic analysis for stochastic QoS of Web service composition. *J. Softw.* **53**(14), 70–75 (2017)
9. Wang, S.G., Sun, Q.B., Zhang, G.W.: Uncertain QoS-aware skyline service selection based on cloud model. *J. Softw.* **23**(6), 1397–1412 (2012)
10. Sun, S.X., Zhao, J.: A decomposition-based approach for service composition with global QoS guarantees. *Inf. Sci.* **199**(15), 38–153 (2012)
11. Surianarayanan, C., Ganapathy, G., Ramasamy, M.S.: An approach for selecting best available services through a new method of decomposing QoS constraints. *Serv. Oriented Comput. Appl.* **9**(2), 107–138 (2014). <https://doi.org/10.1007/s11761-014-0154-x>
12. Guidara, I., Jaouhari, I.A., Guermouche, N.: Dynamic selection for service composition based on temporal and QoS constraints. In: *IEEE International Conference on Services Computing*, pp. 267–274. IEEE (2016)
13. Ren, L.F., Wang, W.J., Xu, X.: Uncertainty-aware adaptive service composition in cloud computing. *Comput. Eng. Appl.* **53**(12), 2867–2881 (2016)
14. Ye, H.Z., Lu, X.P.: Robust web services composition based on discrete particle swarm optimization. *J. Univ. Electron. Sci. Technol. China* **47**(03), 443–448 (2018)
15. Zheng, Z., Zhang, Y., Lyu, M.R.: Investigating QoS of real-world web services. *IEEE Trans. Serv. Comput.* **7**(1), 32–39 (2014)
16. Hwang, S.-Y., Wang, H., Tang, J.: A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. *Inf. Sci.* **177**(23), 5484–5503 (2007)
17. Bai, L., Liang, J., Cao, F.: A multiple k-means clustering ensemble algorithm to find nonlinearly separable clusters. *Inf. Fusion* **61**, 36–47 (2020)



A High-Performance with Low-Resource Utility FPGA Implementation of Variable Size HEVC 2D-DCT Transform

Ying Zhang¹(✉), Gen Li²(✉), and Lei Wang¹(✉)

¹ Lab 673-2, Institute of Computer, National University of Defense Technology,
Changsha 410073, China

{zhangying, wanglei}@nudt.edu.cn

² Genetalks Biotech. Co., Ltd, Changsha 410073, China
gen.li@genetalks.com

Abstract. High Efficiency Video Coding (HEVC) is a new international video compression standard offering much better compression efficiency than previous video compression standards at the expense of much higher computational complexity. This paper presents a design of two-dimensional (2D) discrete cosine transform (DCT) hardware architecture dedicated for High Efficiency Video Coding (HEVC) in field programmable gate array (FPGA) platforms. The proposed methodology efficiently proceeds 2D-DCT computation to fit internal components and characteristics of FPGA resources. This architecture supports variable size of DCT computation, including 4×4 , 8×8 , 16×16 , and 32×32 , and has been implemented in Verilog and synthesized in various FPGA platforms. Compared with existing related works, our proposed architecture demonstrates significant advantages in the performance improvement with low FPGA resource utility, which are very important for the whole FPGA solution for whole HEVC codec.

Keywords: H.265/HEVC · Two-dimensional discrete cosine transform (2D-DCT) · FPGA · Hardware

1 Introduction

Rapid advances in consumer electronics have resulted in a variety of video coding applications, such as ultra-high definition (UHD) 4 K/8 K TV [1] or unmanned aerial vehicle (UAV) reconnaissance and surveillance [2, 3], which demands aggressive video compression requirement. ITU and ISO standardization organizations are jointly developing a new international video compression standard H.265/HEVC, which has great potential to improve video compression efficiency by around 50%, while retaining the same video quality as H.264 [5, 6]. As a result, HEVC has been viewed as one of the most promising standards to overcome these challenges [7, 8]. However, High Efficiency Video Coding (HEVC) video compression standard at the expense of much more computational complexity [4–9]. HEVC uses Discrete Cosine Transform (DCT)/Inverse Discrete Cosine

Transform (IDCT). In addition, it uses Discrete Sine Transform (DST)/Inverse Discrete Sine Transform (IDST) for 4×4 intra prediction in certain cases. Additionally, HEVC supports more block sizes of DCT and IDCT, including 32×32 , 16×16 , 8×8 , and 4×4 than H.264 which supports two smaller block sizes (i.e. 4×4 and 8×8). Large DCT and DST have high computational complexity, and they are heavily used in an HEVC encoder [10]. DCT and DST operations account for 11% of the computational complexity of an HEVC video encoder. They account for 25% of the computational complexity of an all intra HEVC video encoder.

Nowadays, computational resources in FPGA makes the study of HEVC FPGA implementation is gaining more and more attention. An efficient 2-D DCT FPGA architecture, used for the whole HEVC FPGA solution should have the following features: supporting all possible sizes in HEVC transform; using as little as possible FPGA implementation cost, i.e. the FPGA resources; gaining high-performance as possible.

In this paper, an FPGA implementation of HEVC 2-D DCT transform is proposed. Our FPGA architecture focuses on the above features. Compared with existing 2D-DCT FPGA architecture, our design can compute all block sizes in HEVC with the same FPGA design, which means the FPGA resources for butterfly PEs can deal with the block sizes of 32×32 , 16×16 , 8×8 and 4×4 , thus greatly reducing the FPGA implementation cost. Our butterfly PEs, which compute one element for the block size of 32×32 per cycle, can compute 2 elements for the block size of 16×16 per cycle, 4 elements for the block size of 8×8 and 8 elements for the block size of 4×4 per cycle, thus gaining high computation efficiency.

The rest of the paper is organized as follows. In Sect. 2, HEVC transform algorithms and related FPGA solutions for 2-D DCT are explained. In Sect. 3, the proposed element computation of HEVC DCT is explained. In Sect. 4, FPGA architecture for HEVC DCT is described. The implementation results are given in Sect. 5. Finally, Sect. 6 presents the conclusion.

2 Introduction

2.1 HEVC Transform Algorithms

Formula (1) shows the basic function for HEVC 1-D DCT transformation for an $N \times N$ block, where $i, j = 0, \dots, N-1$. HEVC uses 4×4 , 8×8 , 16×16 and 32×32 TU sizes for DCT [11].

$$T_{i,j} = \omega_0 \cdot \sqrt{\frac{2}{N}} \cdot \cos\left(\frac{\pi \cdot i \cdot (2j + 1)}{2N}\right), \omega_0 = \begin{cases} \sqrt{\frac{2}{N}} & i = 0 \\ 1 & i \neq 0 \end{cases} \quad (1)$$

HEVC performs 2D transform operation by applying 1D transforms in vertical and horizontal directions. The coefficients in HEVC. 1D transform matrices are derived from DCT basis functions. However, integer coefficients are used for simplicity. HEVC 1D DCT coefficients for 4×4 TU size are shown in Fig. 1. The coefficients in the same even row are symmetrical, while those in the odd row are odd-symmetrical, i.e. with the same values but different signals. Thus, the butterfly computation is used to compute every element in the DCT transformation.

$$DCT_{4 \times 4} = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix}$$

Fig. 1. HEVC 1D DCT coefficients for 4×4 TU size

2.2 Related FPGA Architectures

In order to satisfy real-time and high-efficiency coding in these emerging video applications, a few design methodologies and circuit architectures have been developed [11, 18, 19, 20-24]. Usually, they can be partitioned 3 kinds. [12-15] can only compute one or two TU sizes of DCT transformation. The other TU sizes cannot be processed in these FPGA solutions. These architectures are meaningful for research but cannot be used in the really whole HEVC FPGA system solutions. [16, 17] support all block sizes of HEVC, but they build one different FPGA logics. Thus, while the FPGA logics for one block size work, the other three parts of FPGA resources idle. In [18], the FPGA architecture can support all HEVC block sizes, but all PEs computing one element per cycle for 32×32 block size can only compute one element for the other smaller sizes. There are also some other FPGA designs with their own features, such as comparing the FPGA implementation of multiplications with LUT or DSP [19], using shift-adds instead of multiplications [12], and so on.

3 Proposed FPGA Architecture for Element Computation of HEVC DCT

3.1 FPGA Design for an Element in 32×32 Block Size

FPGA design for an element in 32×32 block size is explained first in this subsection.

Butterfly transform can be applied several times in DCT. Each time an even data matrix splits into smaller even and odd parts, until down to 4×4 size. Even parts may be reused for different DCT sizes, but odd parts are prohibited. Table 1 illustrates how hardware resources for processing one pixel point varies with depth of butterfly transform. It is apparent that more levels of butterfly transforms require less number of multipliers.

We propose to apply butterfly transform only once, instead of three times as in [6]. With the resource overhead of six more multipliers, the benefit of our design is to reuse these multipliers, which need a lot of FPGA resources, in smaller DCT sizes. Thus, the multipliers in our design can compute the multiplication for 32×32 , 16×16 , 8×8 and 4×4 block sizes, hence greatly improving the efficiency of the FPGA resources, which is greatly important for the whole system of HEVC FPGA solution.

Figure 2 shows the hardware design for an element computation by a row and an even coefficient column, in a 32×32 block size. The result of residual data R0 plus R31 and the first coefficient C0 are supplied to the multiplier MUTI0; the result of residual data R1 plus R30 and the first coefficient C1 are supplied to the multiplier MUTI1, and so on.

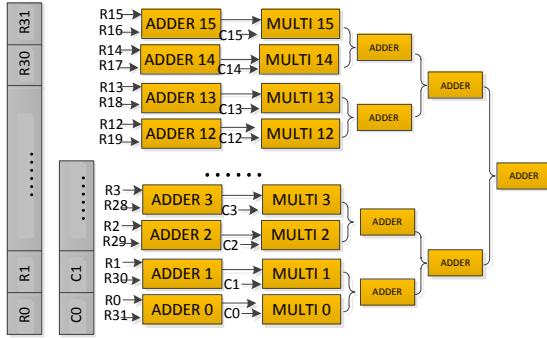


Fig. 2. An element computation by a row and an even coefficient column in a 32×32 block size

Then the results of MULTI0, MULTI1, ..., and MULTI15 are accumulated together. Thus, an element for 1D-DCT is generated.

With all adders for butterfly computation replaced with subtractors, the hardware design above can generate an element for a row with an odd column coefficient data.

3.2 FPGA Architecture for an Element in Other Block Sizes

Then the computation of the element in 16×16 Block Size by the Multiplier and Adder/subtractor in Fig. 1 is explained. While 16 multipliers and adders are used to compute an element in 32×32 block size, only half of them are needed to compute an element in 16×16 block size. To gain high efficiency of the FPGA resources, especially the multipliers which cost high FPGA resources, we compute 2 elements once using the logic in Fig. 1. Thus, all 16 multipliers and adders work at the same time. Figure 3 gives the computation of the elements in 16×16 Block Size.

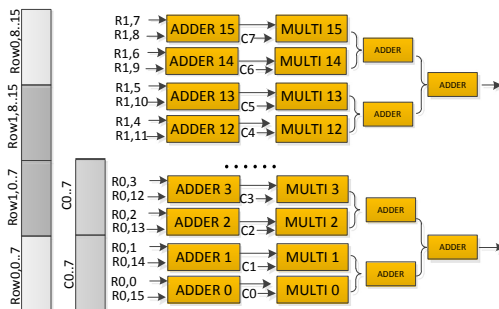


Fig. 3. Two elements generated by two rows and an even coefficient column in a 16×16 block size

As shown in Fig. 3, two rows, $r_{0,0}, \dots, r_{0,15}$ and $r_{1,0}, \dots, r_{1,15}$ are organized as one 32-length row, $r_{0,0}, \dots, r_{0,7}, r_{1,0}, \dots, r_{1,7}, r_{1,8}, \dots, r_{1,15}, r_{0,8}, \dots, r_{0,15}$. The 8-length coefficient column is organized as one 16-length coefficient column data, $c_0, \dots, c_7, c_0, \dots, c_7$.

Then the organized 32-length row and the 16-length coefficient column are inputted to the multipliers and adders for element computation in 32×32 block size. Two elements are outputted by the adders before the last level. With all adders for butterfly computation replaced with subtractors, the hardware design above can generate two elements for a row with an odd column coefficient data.

As analyzed above, four elements of 8×8 block size can be generated once by the same FPGA multipliers and adders with four rows and a coefficient column; 8 elements of 4×4 block size can be generated once by the same FPGA multipliers and adders with 8 rows and a coefficient column. Figure 4 shows the data organization of the residual rows and coefficient columns for 8×8 block size. For 4×4 block size, 8 elements, i.e. two rows, are generated once, with similar data organization.

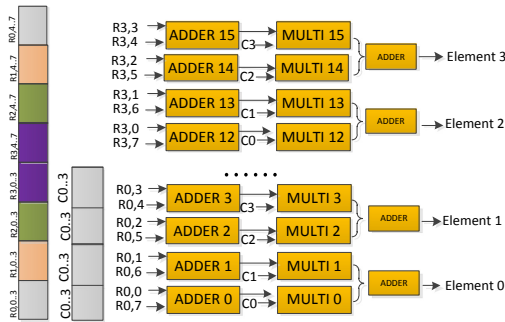


Fig. 4. Four elements generated by four rows and an even coefficient column in an $8 \times$ block size

As shown in Fig. 2 and Fig. 3, the same computation FPGA logic can compute one 32×32 block size element or two 16×16 block size elements once.

To increase hardware utilization efficiency, on-chip DSP blocks are preferred to realize multiplication-addition. Here we propose to map butterfly transform into DSP48 blocks in Xilinx xc7vx690 FPGA.

4 FPGA Architecture for HEVC DCT

The proposed HEVC 2D-DCT transform hardware for all variable block sizes is shown in Fig. 5. The proposed hardware performs 2D DCT by first performing 1D DCT on the rows of a TU, then reordering the intermediate results to the transpose ram, and performing 1D DCT on the columns of the TU. The resulting transformed coefficients, generated by 1D row DCT, are reordered from row-sequence to row-sequence and stored in a transpose memory, and they are used as input for 1D column DCT. The FPGA architecture for 1D row DCT and 1D column DCT is the same, so only 1D row DCT is described below.

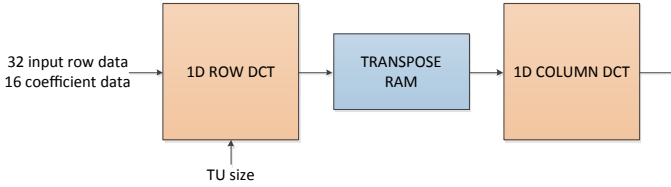


Fig. 5. Hardware design for HEVC 2D-DCT

Figure 6 shows our algorithm and the overall architecture. The 1D-DCT FPGA design includes two even FPGA design shown in Figs. 2, 3, or 4, and two odd FPGA design. Thus, our proposed HEVC design transforms 2 32-length rows and 2 16-length coefficient columns at the same time. Either $2 \times$ block for 32×32 block size, 4×2 block for 16×16 block size, 8×2 block for 8×8 block size, or two 4×4 blocks for 4×4 block size, is generated per cycle.

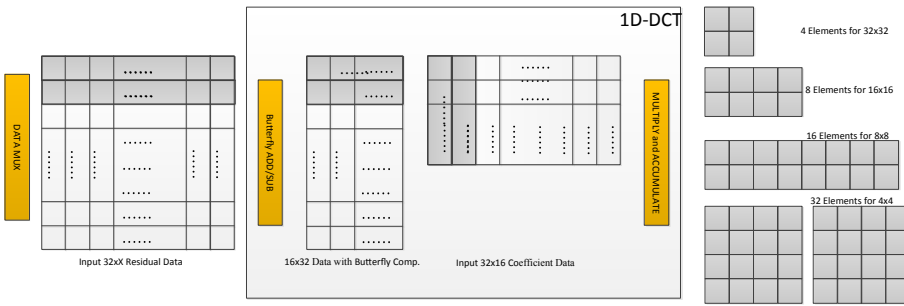


Fig. 6. Proposed algorithm and the overall architecture

Figure 7 shows our proposed architecture for HEVC 1D-DCT. The residual data row is organized as described above according to the TU size. The data mux/organization unit processes the data organization. The organized data is input to 1-D DCT unit, which generates the 1D-DCT coefficients. Finally, the 1D-DCT coefficients are reordered and written to the transpose ram. There are 3 stages in our FPGA design, one for data mux unit and two for 1D-DCT transformation.

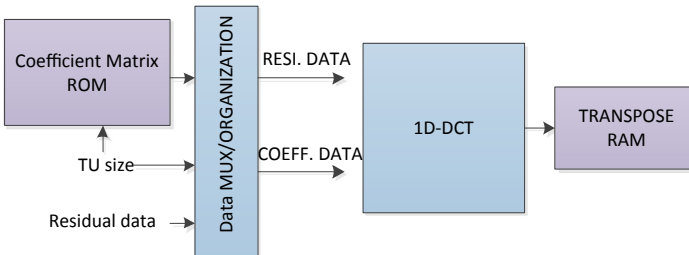


Fig. 7. FPGA architecture for HEVC 1D-DCT

5 Experiments

The proposed FPGA architecture for HEVC 2D-DCT transform is implemented using Verilog HDL. The Verilog RTL codes are verified with RTL simulations. RTL simulation results matched results of HEVC 2D-DCT transform implementation in Joint Exploration Test Model (JEM) 4.0 reference software encoder [2]. The Verilog RTL codes are synthesized and mapped to a Xilinx XC7VX690TFFG761-2 FPGA with speed grade 2 using Xilinx ISE. The FPGA implementation is verified to work at 250 MHz by post place and route simulations. Table 1 shows the FPGA resources needed by our design.

Table 1. FPGA resources needed by our design

Resource	Utilization	Available	Utilization %
LUT	2900	433200	0.67
FF	302	866400	0.03
BRAM	16	1470	1.09
DSP	64	3600	1.78

As shown in Table 1, our design has low cost of the FPGA resources.

Our design also has high computation efficiency. Table 2 compares the cycles needed for each block size 1D-DCT transformation, with [18] which proposes a design aimed both performance and the FPGA cost.

Table 2. Comparison of the cycles needed for each block size 1D-DCT transformation, with [18]

	Our design	Design in [18]
32×32	256	256
16×16	32	64
8×8	4	16
4×4	1/2	4

As shown in Table 2, compared with [18], our proposed FPGA design have much higher performance for little block sizes.

As demonstrated in our experiments, our FPGA design can support all possible sizes in HEVC transform, use as little as possible FPGA implementation cost, i.e. the FPGA resources, and gain high-performance as possible. Therefore, our FPGA architecture can be used for the whole HEVC FPGA solutions as the 2-D DCT hardware design.

6 Conclusion

Nowadays, computational resources in FPGA makes the study of whole HEVC FPGA implementation is gaining more and more attention. An efficient 2-D DCT FPGA architecture, used for the whole HEVC FPGA solution should have the following features: supporting all possible sizes in HEVC transform; using as little as possible FPGA implementation cost, i.e. the FPGA resources; gaining high-performance as possible. Compared with existing 2D-DCT FPGA architecture, our design can compute all block sizes in HEVC with the same FPGA design, which means the FPGA resources for butterfly PEs can deal with the block sizes of 32×32 , 16×16 , 8×8 and 4×4 , thus greatly reducing the FPGA implementation cost. Our butterfly PEs, which compute one element for the block size of 32×32 per cycle, can compute 2 elements for the block size of 16×16 per cycle, 4 elements for the block size of 8×8 and 8 elements for the block size of 4×4 per cycle, thus gaining high computation efficiency.

Acknowledgments. We acknowledge the reviewers for their insightful comments. This work is supported by the HGJ2017 under Grant No.2017ZX01028103 and Grant 2018ZX01029103.

References

1. Meuel, H., Munderloh, M., Ostermann, J.: Stereo mosaicking and 3D video for single-view HDTV aerial sequences using a low bit rate ROI coding framework. In: International Conference on Advanced Video and Signal Based Surveillance, pp. 1–6 (2015)
2. Bhaskaranand, M., Gibson, J.: Low-complexity video encoding for UAV reconnaissance and surveillance. In: Military Communications Conference, pp. 1633–1638 (2011)
3. Bhaskaranand, M., Gibson, J.: Low complexity video encoding and high complexity decoding for UAV reconnaissance and surveillance. In: International Symposium on Multimedia, pp. 163–170 (2013)
4. Zhang, Q., Chang, H., Huang, X., Huang, L., Su, R., Gan, Y.: Adaptive early termination mode decision for 3D-HEVC using inter-view and spatio-temporal correlations. *Int. J. Electron. Commun.* **70**(5), 727–737 (2016)
5. Bossen, F., Bross, B., Suhring, K., Flynn, D.: HEVC complexity and implementation analysis. *IEEE Trans. Circuits Syst. Video Technol.* **22**(12), 1685–1696 (2012)
6. Kalali, E., Ozcan, E., Yalcinkaya, O., Hamzaoglu, I.: A low energy HEVC inverse transform hardware. *IEEE Trans. Consum. Electron.* **60**(4), 754–761 (2014)
7. Kessentini, A., Samet, A., Ayed, M., Masmoudi, N.: Performance analysis of inter-layer prediction module for H.264/SVC. *Int. J. Electron. Commun.* **69**(1), 344–350 (2015)
8. Samcovic, A.: Mathematical modeling of coding gain and rate-distortion function in multi-hypothesis motion compensation for video signals. *Int. J. Electron. Commun.* **69**(2), 487–491 (2015)
9. Budagavi, M., Fuldseth, A., Bjontegaard, G., Sze, V., Sadafale, M.: Core transform design in the high efficiency video coding (HEVC) standard. *IEEE J. Selected Topics Signal Process.* **7**(6), 1029–1041 (2013)
10. Rao, K.R., Yip, P.: Discrete Cosine Transform: Algorithms, Advantages, Applications. Academic Press, Inc., Cambridge (1990)
11. Kalali, E., Mert, A.C., Hamzaoglu, I.: A computation and energy reduction technique for HEVC discrete cosine transform. *IEEE Trans. Consum. Electron.* **62**(2), 166–174 (2016)

12. Conceicao, R., Souza, J., Jeske, R., Zatt, B., Porto, M., Agostini, L.: Low-cost and high throughput hardware design for the HEVC 16×16 2-D DCT transform. *J. Integr. Circ. Syst.* **9**, 25–35 (2014)
13. Mert, A.C., Kalali, E., Hamzaoglu, I.: An FPGA implementation of future video coding 2D transform. In: *IEEE International Conference on Consumer Electronics - Berlin*. IEEE (2017)
14. Park, J.S., Nam, W.J., Han, S.M., et al.: 2-D large inverse transform (16×16 , 32×32) for HEVC (High Efficiency Video Coding). *J. Semicond. Technol. Sci.* **12**(2), 203–211 (2012)
15. Chen, T.H.: A cost-effective 8×8 2-D IDCT core processor with folded architecture. *IEEE Trans. Consum. Electron.* **45**(2), 333–339 (1999)
16. Sjovall, P., Viitamaki, V., Vanne, J., et al.: High-level synthesis implementation of HEVC 2-D DCT/DST on FPGA. In: *IEEE International Conference on Acoustics*. IEEE (2017)
17. Huang, J., Parris, M., Lee, J., et al.: Scalable FPGA architecture for DCT computation using dynamic partial reconfiguration. *ACM Trans. Embedded Comput. Syst.* **9**(1), 269–272 (2009)
18. Chen, M., Zhang, Y., Lu, C.: Efficient architecture of variable size HEVC 2D-DCT for FPGA platforms. *Aeu Int. J. Electron. Commun.* **73**, 1–8 (2017)
19. Mert, A.C., Kalali, E., Hamzaoglu, I.: An FPGA implementation of future video coding 2D transform. In: *IEEE International Conference on Consumer Electronics - Berlin*. IEEE (2017)

Author Index

- Cai, Jinyan 221
Cao, Jijun 16, 56
Cao, Qiang 163
Chang, Junsheng 56
Chen, Juan 135, 233
Chen, Qiurui 192
- Dai, Yi 16, 56
Deng, Liang 260
Ding, Dong 117
Dong, Yong 135, 233
- Fang, Jianbin 150
Feng, Quanyou 209
Fu, ZhiPeng 249
- Gao, Wanrong 150
Gao, Wenqiang 178
Gao, Yin 304
Guan, Yijin 73
Guo, Kaile 31
- Han, Yinhe 73
Huang, Chun 150
- Ji, Weixing 192
Jin, Kang 43
- Kang, Ziyang 87
- Lai, Mingche 16, 31, 56
Li, Cunlu 43
Li, Gen 325
Li, Hongyun 304
Li, Jun 304
Li, Qiong 178
Li, Shiming 87, 277
Liang, Dongbao 101
Lin, Bai 3
Lin, Rong-Fen 290
Liu, Tao 31
- Liu, Yongheng 249
Lou, Hui 43
Lu, Dechao 31
Lu, Pingjing 56
Lu, Zhang 315
Luo, Li 117, 209
Lv, Fangxu 31
- Meng, Dehong 260
- Niu, Dimin 73
- Pan, Guoteng 209
Pang, Ling 3
Pang, Zhengbin 16, 31
- Qi, Feng-Bin 290
Qi, Xingyun 16, 56
Qi, Xinxin 233
Qu, Lianhua 87
- Su, Jinshu 87
Su, Tao 101
Su, Yijing 304
Sun, Guangyu 73
Sun, Xiaole 135
Sun, Yan 260
- Tang, Weiping 31
- Wang, Fei 290
Wang, Lei 87, 117, 277, 325
Wang, Shiyang 87, 277
Wang, Shucheng 163
Wang, Yizhuo 192
Wang, Yuhao 73
Wang, Yuntao 260
Wang, Zhao 73
Wang, Zheng 135
Wei, Dengping 178
Wei, Shuangjian 192

Wu, Miaomiao 31
Wu, Wei 290
Wu, Yuxuan 31

Xiao, Canwen 43
Xiao, Jiale 101
Xie, Chuan 117
Xie, Xuchao 178
Xu, Chuanfu 150
Xu, Weixia 277

Yang, Shazhou 249
Yang, Zhijie 117
Ye, Zhixia 221
Yu, Yangbin 101

Yuan, Lingyun 221
Yuan, Yuan 233
Yue, Hao 260

Zeng, Rui 221
Zhang, Xiangyu 117
Zhang, Ying 325
Zhao, Jinjing 3
Zhao, Longfei 249
Zheng, Hongzhong 73
Zhou, Hailiang 209
Zhou, Li 209
Zhou, Ye Heng 315
Zhu, Qi 290
Zhu, Yue 249