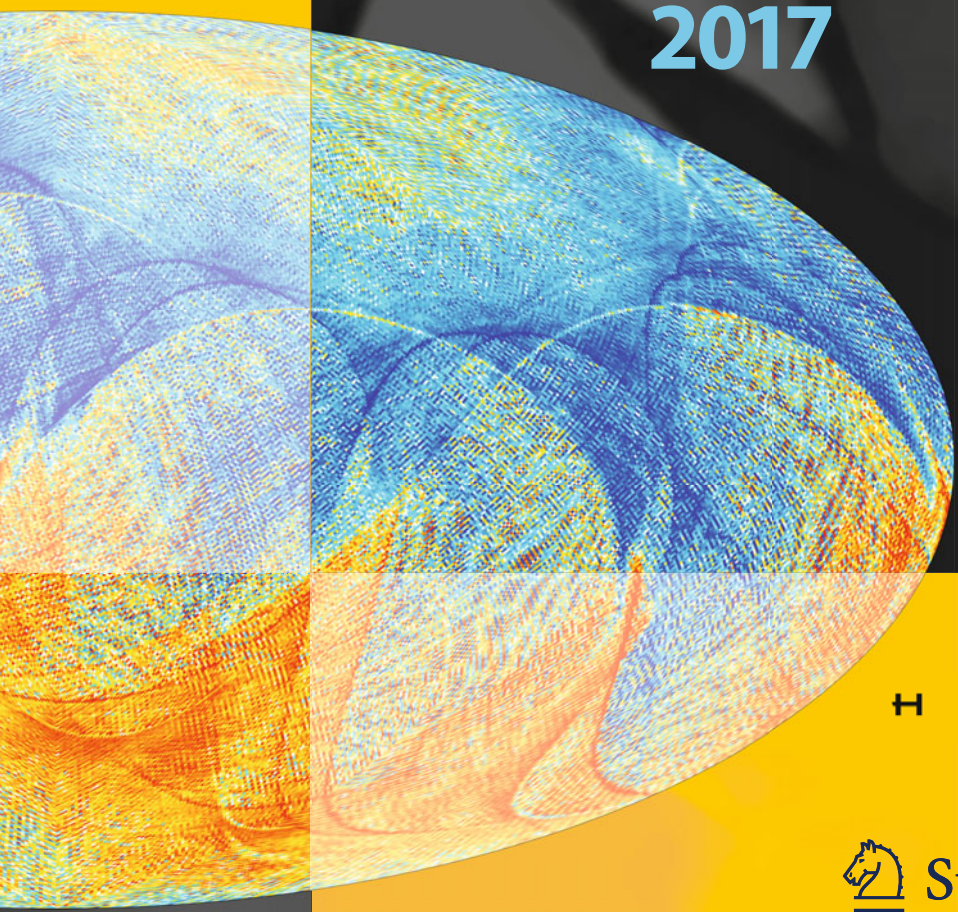
Christoph Niethammer · Michael M. Resch
Wolfgang E. Nagel · Holger Brunst
Hartmut Mix
Editors

Tools for High Performance Computing 2017



HLRS

Springer

Tools for High Performance Computing 2017

Christoph Niethammer · Michael M. Resch ·
Wolfgang E. Nagel · Holger Brunst ·
Hartmut Mix
Editors

Tools for High Performance Computing 2017

Proceedings of the 11th International Workshop on Parallel Tools for High Performance Computing, September 2017, Dresden, Germany

Springer

Editors
Christoph Niethammer
Höchstleistungsrechenzentrum
Stuttgart (HLRS)
Universität Stuttgart
Stuttgart, Germany

Michael M. Resch
Höchstleistungsrechenzentrum
Stuttgart (HLRS)
Universität Stuttgart
Stuttgart, Germany

Wolfgang E. Nagel
Zentrum für Informationsdienste
und Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden, Germany

Holger Brunst
Zentrum für Informationsdienste
und Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden, Germany

Hartmut Mix
Zentrum für Informationsdienste
und Hochleistungsrechnen (ZIH)
Technische Universität Dresden
Dresden, Germany

ISBN 978-3-030-11986-7 ISBN 978-3-030-11987-4 (eBook)
https://doi.org/10.1007/978-3-030-11987-4

Library of Congress Control Number: 2019930155

Mathematics Subject Classification (2010): 68M14, 68M20, 68Q85, 65Y05, 65Y20

Cover illustration: Sky map of systematic errors in one component of the celestial position in the star catalog of a simulated ESA Gaia mission if thermo-mechanic instabilities are not calibrated. The calibration of the real Gaia data for such effects is using a significant amount of CPU time on the HPC system of the Center for Information Services and High Performance Computing (ZIH). Data: S. Klioner, Visualization: R. Geyer, Lohrmann-Observatorium—TU Dresden, Gaia DPAC.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The advances in High Performance Computing are today forced by an impressive growth of the computer systems. This is connected with a drastic increasing number of compute elements. At the same time, the systems also become more and more complex, because of the combination of different special designed and optimized processors or accelerators. These advances steadily allow to calculate more complex or even new scientific or technical problems.

This progress is also connected with an even more increasing of the complexity of the used software applications. This demands a challenging effort from the software developers to utilize the entire potential of these hardware systems. Here, powerful software analysis and optimization tools are required that support application developers during the software design, implementation, and testing process. As a result, many international research groups are specialized in the development and improvement of such software tools. They all share the view that their tools could deliver an important contribution to the progress in the computer usage.

The International Parallel Tools Workshop is a series of workshops that already started in 2007 at the High Performance Computing Center Stuttgart (HLRS) and currently takes place once a year. The goal of these workshops is to bring together HPC tools developers and users from science and industry to learn about new achievements and to discuss future development approaches. The scope includes HPC-related tools for performance analysis, debugging, or system utilities as well as presentations providing feedback and experiences from tool users.

The 11th International Parallel Tools Workshop[1] took place on September 11–12, 2017 in Dresden, Germany. In the presentations and discussions during this workshop, both aspects of the tools advancement were addressed. New tools developments have been described, and also examples for successful usage of them to analyze and optimize HPC applications have been presented. Main research topics shown at this workshop were structured performance analysis approaches, unified and online instrumentation, profiling and optimization assistance, and the investigation of the resource management. A presented counter inspection toolkit

[1] https://tools.zih.tu-dresden.de/2017/.

could assist application developers in investigating and categorizing the increasing number and complexity of hardware counters that are offered by the vendors of modern hardware. The content of the presentations comprised a broad spectrum of modern programming techniques. Task-based programming and OpenMP execution models were included as well as development assistance for accelerator-supported applications.

The book contains the contributed papers to the presentations held on the workshop in September 2017 in Dresden. As in the previous years, the workshop was jointly organized by the Center of Information Services and High Performance Computing (ZIH)[2] of the Technische Universitaet Dresden and the High Performance Computing Center Stuttgart (HLRS).[3]

Dresden, Germany

Hartmut Mix
Holger Brunst
Christoph Niethammer
Michael M. Resch
Wolfgang E. Nagel

[2] https://tu-dresden.de/zih.

[3] https://www.hlrs.de.

Contents

A Structured Approach to Performance Analysis 1
Michael Wagner, Stephan Mohr, Judit Giménez and Jesús Labarta

Counter Inspection Toolkit: Making Sense Out of Hardware Performance Events 17
Anthony Danalis, Heike Jagode, Hanumantharayappa, Sangamesh Ragate and Jack Dongarra

ASSIST: An FDO Source-to-Source Transformation Tool for HPC Applications 39
Youenn Lebras, Andres S. Charif Rubial, Romain Dolbeau and William Jalby

Unifying the Analysis of Performance Event Streams at the Consumer Interface Level 57
Jean-Baptiste Besnard, Allen D. Malony, Sameer Shende, Marc Pérache, Patrick Carribault and Julien Jaeger

OMPT-Multiplex: Nesting of OMPT Tools 73
Joachim Protze, Tim Cramer, Simon Convent and Matthias S. Müller

SCIPHI Score-P and Cube Extensions for Intel Phi 85
Marc Schlütter, Christian Feld, Pavel Saviankou, Michael Knobloch, Marc-André Hermanns and Bernd Mohr

Towards Elastic Resource Management 105
Isaías A. Comprés Ureña and Michael Gerndt

Online Performance Analysis with the Vampir Tool Set 129
Matthias Weber, Johannes Ziegenbalg and Bert Wesarg

A Structured Approach to Performance Analysis

Michael Wagner, Stephan Mohr, Judit Giménez and Jesús Labarta

Abstract Performance analysis tools are essential in the process of understanding application behavior, identifying critical performance issues and adapting applications to new architectures and increasingly scaling HPC systems. State-of-the-art tools provide extensive functionality and a plenitude of specialized analysis capabilities. At the same time, the complexity of the potential performance issues and sometimes the tools themselves remains a challenging task, especially for non-experts. In particular, identifying the main issues in the overwhelming amount of data and tool opportunities as well as quantifying their impact and potential for improvement can be tedious and time consuming. In this paper we present a structured approach to performance analysis used within the EU Centre of Excellence for Performance Optimization and Productivity (POP). The structured approach features a method to get a general overview, determine the focus of the analysis, and identify the main issues and areas for potential improvement with a statistical performance model that leads to starting points for a subsequent in-depth analysis. All steps of the structured approach are accompanied with according tools from the BSC tool suite and underlined with an exemplary performance analysis.

1 Introduction

Taking advantage of the full computational potential of contemporary high performance computing (HPC) systems is a challenging and intricate task. It necessitates not only considering parallel execution, network, system topology, and hardware accelerators but also the proper usage of a variety of different parallel programming models such as message passing, threading and tasking, one-sided communication, or architecture specific models to include hardware accelerators.

M. Wagner (✉) · S. Mohr · J. Giménez · J. Labarta
Barcelona Supercomputing Center (BSC), C/ Jordi Girona, 29, 08034 Barcelona, Spain
e-mail: michael.wagner@bsc.es

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_1

To ease the complexity of the continuing optimization and adaption of software, performance analysis tools support developers not only in identifying performance issues within their applications but also in understanding how new architectures and increasingly scaling HPC systems affect their parallel behavior. While performance analysis tools are steadily improving their functionality and usability, the complexity of the parallel behavior, convoluted and multifaceted causes of performance issues, and sometimes the tools themselves pose a challenging and often overwhelming task, especially, for non-experts.

In recent years it is generally recognized that this burden cannot and should not be put on domain scientists but must be carried by experts from computer science. This is reflected among others in the promotion of co-design efforts and lately in the creation of the EU Centre of Excellence for Performance Optimization and Productivity (POP) [1]. POP targets the frequent lack of quantified understanding of actual behavior and missing knowledge of the most productive direction of code refactoring by providing performance analysis as a service to code developers, software users, and infrastructure operators.

But even for experts, identifying the main issues in the overwhelming amount of data and tool opportunities as well as quantifying their impact and potential for improvement can be tedious and time consuming at least. An unstructured approach to performance may overlook certain behavioral aspects or performance issues. At the end, the goal of performance analysis is not only to identify and solve some of the performance issues but rather understanding the application behavior profoundly, quantifying the achieved performance and improvement potential, and guiding to the most productive development efforts. Furthermore, the performance analysis itself should be carried out in the most productive way to avoid wasting time and resource. This becomes particularly important when the performance analysis is executed as a service, as in the EU Centre of Excellence for Performance Optimization and Productivity.

In this paper, we propose a structured approach to performance analysis established for many years at BSC as well as the EU Centre of Excellence for Performance Optimization and Productivity. The structured approach incorporates five steps from collecting a representative measurement set, to getting a general overview and defining the focus of analysis, identifying the main issues and areas for potential improvement with a statistical performance model, an in-depth analysis guided by the performance model, and, finally, a documenting and reporting the analyzed behavior, performance issues and recommendations.

In the remainder of the paper, we discuss the necessary background to better follow the argumentation in Sect. 2. After that, we present the structured approach to performance analysis and discuss it along an exemplary performance analysis from POP in Sect. 3. Finally, Sect. 4 summarizes the work and draws conclusions.

2 Background

This section introduces the EU Centre of Excellence for Performance Optimization and Productivity (POP), the BSC tools, and the example code CheSS used for the analysis in the following chapter.

2.1 The EU Centre of Excellence for Performance Optimization and Productivity (POP)

The key motivation for the Centre of Excellence in Performance Optimization and Productivity is advancing the productivity of EU research and industry by providing free of charge services that help improving the performance of high performance computing and parallel software.

POP consists of six European partners: Barcelona Supercomputer Center (BSC), High Performance Computing Center Stuttgart (HLRS), Juelich Supercomputing Centre (JSC), Numerical Algorithms Group (NAG), RWTH Aachen and TERATEC. All partners are characterized by excellence in performance analysis and tuning, best practices in programming models and parallel programming, a strong research and development background, as well as a proven commitment in their application to real academic and industrial use cases.

POP offers a portfolio of services designed to help users optimize parallel software, identify and understand potential performance issues. While the primary customers are code developers, the services are also suited for code users and infrastructure providers. The central services are: Performance Audits, Performance Plans, Proof-of-Concept implementations, and Training.

Performance Audits provide an initial analysis and overview that measures a range of performance metrics to assess quality of performance and identify the issues affecting performance. Performance Plans undertake further performance evaluations to qualify and quantify solutions and estimate potential improvements. Proof-of-Concept implementations use extracted application kernels to demonstrate actual benefits from tuning and optimization. Training offers events and materials covering parallel profiling tools, programming models and parallelization approaches.

2.2 The BSC Tools

For our performance measurements we choose the open source BSC tools [2] including the Extrae trace monitor [3] and the Paraver trace analyzer [4] for two main reasons. First, Extrae combines the benefits of instrumentation and sampling by intercepting the parallel runtime to provide the exact communication behavior but uses sampling instead of function instrumentation to record the application behavior

in the compute phases. Second and for this case most important, Extrae and Paraver support all steps with the according functionality.

The central analysis tool is Paraver, a trace-based performance analyzer featuring great flexibility to explore and extract information. Paraver provides two main visualizations: timelines, which graphically display the evolution of application behavior over time, and tables (profiles and histograms), which deliver statistical information. These two complementary views allow easy identification of computational inefficiencies, load balancing issues, serializations that affect the parallel scalability, cache and memory impact on the performance, and regions with generally low performance. In addition, Paraver contains multiple analytic modules that allow, for instance, the automatic generation of key performance indicators and performance quantification (basic analysis), the semi-automatic detection of application structure (clustering), and the tracking of key performance characteristics over multiple measurements to identify scalability limitation (tracking).

The Dimemas simulator allows a fast evaluation of what-if scenarios for MPI applications, for example, to evaluate the benefits of moving to a machine with a faster network, or potential improvements from better load balance.

The performance data collection is performed by the Extrae trace monitor. Extrae intercepts the common parallel runtime environments (MPI, OpenMP, OmpSs, Pthreads, CUDA, OpenCL, SHMEM) and supports all major programming languages (C, C++, Fortran, Python, JAVA). It has been successfully ported to a wide range of platforms like Intel, Cray, BlueGene, Fujitsu Spark, MIC, and ARM. On most platforms the preload mechanism allows avoiding specific compilations and working with the unmodified production binary.

2.3 Example Code: CheSS

In order to simulate the properties of matter at a microscopic scale on has to explicitly take into account the atomic structure of the system under investigation. Within these so-called atomistic simulations there exist many different levels of description, ranging from empirical approaches yielding a moderate precision to very accurate ab-initio methods that are based on the fundamental laws of Quantum Mechanics.

During the past years, Density Functional Theory (DFT) has established itself as the de facto standard for atomistic simulations on an ab-initio level thanks to its good compromise between accuracy and speed. The main advantage of DFT is that it only scales cubically with respect to the system size—this might sound bad, but is actually much better than that of other ab-initio methods. Therefore, DFT allows performing calculations up to a few hundred atoms while keeping an accurate ab-initio description. Beyond these sizes the numerical cost becomes too high due to the cubic scaling. However, there exist also linear scaling implementations of DFT that allow accessing larger sizes, and indeed with such approaches calculations up to thousands of atoms are possible.

Nowadays there exist many different implementations of DFT in numerous scientific codes, which mainly differ in the basis set that they use. We focus in the following on localized basis sets, which are a necessary requirement for large scale DFT codes such as SIESTA, BigDFT, Quickstep, ONETEP, and Conquest. However, due to their localized character these various basis sets still have an important point in common, namely that they lead to sparse matrices. Even more important, the tasks to be performed with these sparse matrices are—once they are calculated—basically identical for all codes. Moreover, in order to reach optimal performance, it is important to exploit the sparsity of the matrices as much as possible.

These considerations have led to the creation of the *CheSS* library [5]. CheSS can perform the required matrix operations that are necessary for a DFT code working with set of localized basis, taking into account the sparsity of the matrices. The algorithm of CheSS is based on an expansion in terms of Chebyshev polynomials. Apart from their optimal interpolation properties, Chebyshev polynomials have the advantage that they can be calculated with a simple recursion formula that allows a very efficient parallelization. More precisely, the Chebyshev approach makes it possible to calculate the individual columns of the final matrix basically independently from each other. Only at the very end a global communication is necessary to gather the individually calculated columns. In CheSS we exploit this feature by parallelizing the calculation of the columns using distributed memory parallelization (MPI). On a second level we add a shared memory parallelization (OpenMP) to further improve the parallel scalability of the library.

The most intensive task of CheSS consists in repeated matrix vector operations to build up the columns of the Chebyshev matrix polynomials. Unfortunately, this is an operation with a rather bad ratio of computation to memory access, and obtaining a parallelization of this part is therefore quite difficult. Nevertheless, much effort was invested into an efficient parallelization to reach a good scalability of CheSS [5].

As a stand-alone library, CheSS intends to be as independent as possible of the DFT code that interfaces it. The only input that it requires is the sparse matrices that it shall process. At present CheSS has been coupled to the two DFT codes SIESTA and BigDFT. Whereas BigDFT is a relatively young code, SIESTA is one of the most popular DFT codes with a long history and a large user base. This importance is also reflected by the fact that SIESTA is one of the flagship codes of the European Centre of Excellence "Materials design at the Exascale" (MaX) [6].

For the analysis of CheSS within the POP project we used as example an FOE calculation, as explained in detail in Ref. [5]. In a nutshell, FOE takes as input two sparse matrices $\mathbf{H}$ and $\mathbf{S}$ and calculates the sparse matrix $\mathbf{K}$ using the aforementioned Chebyshev expansion. For the matrix-vector multiplications required for the calculations of the Chebyshev polynomials it is furthermore necessary to introduce an additional intermediate sparsity pattern, which we will denote by $\tilde{\mathbf{K}}$. As specific example we took a set of matrices stemming from a calculation with the code BigDFT, namely an ensemble of stacked pentacene molecules. In total the system contained 6876 atoms and 19482 localized basis functions centered on the atoms. The resulting matrices had a degree of sparsity of 98.96% for $\mathbf{S}$, 97.11% for $\mathbf{H}$, 94.30% for $\mathbf{K}$ and 90.85% for $\tilde{\mathbf{K}}$.

3 Structured Performance Analysis

Analyzing the performance of an application poses a significant challenge—not solely for non-experts—and more often than not, performance analysis deteriorates to simply finding "something" that can be improved: some unbalanced parallel behavior, too much time in communication, low compute performance or a function that consumes more time than expected.In recent years it is generally recognized that this burden cannot and should not be put on domain scientists but must be carried by experts from computer science. This is reflected among others in the promotion of co-design efforts and lately in the creation of the EU Centre of Excellence for Performance Optimization and Productivity (POP) [1]. POP targets the frequent lack of quantified understanding of actual behavior and missing knowledge of the most productive direction of code refactoring by providing performance analysis as a service to code developers, software users, and infrastructure operators.

But even for experts, identifying the main issues in the overwhelming amount of data and tool opportunities as well as quantifying their impact and potential for improvement can be tedious and time consuming at least. An unstructured approach to performance may overlook certain behavioral aspects or performance issues. At the end, the goal of performance analysis is not only to identify and solve some of the performance issues but rather understanding the application behavior profoundly, quantifying the achieved performance and improvement potential, and guiding to the most productive development efforts. Furthermore, the performance analysis itself should be carried out in the most productive way to avoid wasting time and resource. This becomes particularly important when the performance analysis is executed as a service, as in the EU Centre of Excellence for Performance Optimization and Productivity.

In this paper, we propose a structured approach to performance analysis established for many years at BSC as well as the EU Centre of Excellence for Performance Optimization and Productivity. The structured approach establishes five steps from collecting a representative measurement set, to getting a general overview and defining the focus of analysis, identifying the main issues and areas for potential improvement with a statistical performance model, an in-depth analysis guided by the performance model, and, finally, a documenting and reporting the analyzed behavior, performance issues and recommendations. While finding things to improve is certainly an appropriate goal, performance analysis, can and should be more than that. First and foremost, performance analysis must provide better understanding of the behavior of an application because lacking quantified and profound understanding of the actual behavior will complicate or even inhibit any optimization attempts. Second, performance analysis is not about finding just "some performance issue" but rather identifying all of the most severe issues. Essential for that is to *quantify* performance to allow to rank performance issue one their impact as well as giving estimates on potential performance gains. Finally, performance analysis must combine both, the quantification of performance and the understanding of application behavior, to provide guidance for optimization and refactoring of the application. Incidentally,

the performance analysis itself should be executed in the most productive way to avoid wasting time and resource.

To achieve this, we propose a structured approach to performance analysis established for many years at teh Barcelona Supercomputing Center as well as the EU Centre of Excellence for Performance Optimization and Productivity. We propose an approach that follows five main steps:

Measurement Collecting a representative set of three to five measurements, e.g., with increasing core counts for a scalability-focused analysis. A set of measurements allows to better understand the evolution of key performance metrics and distinguish between general behavior and behavior specif for a certain number of cores or a certain input.

Overview and focus of analysis Getting an initial overview of the application behavior, detecting the overall structure (e.g. how many repeating phases, iterations etc.). Based on this, selecting the *focus of analysis* (FOA), which contains a application phase with representing behavior, e.g., one or a few iterations. The focus of analysis allows to narrow down the further analysis, make it more comparable between the different measurements of the set by removing, e.g., constant initialization time, and reducing the overall analysis effort.

Performance modeling Using a performance model to determine the performance, efficiency, and evolution of key performance indicators. Our proposed performance model considers among others the parallel efficiency, load balance, communication efficiency, and computation scalability (see Sect. 3.3). Its main target is to quantify performance, identify performance issues, rate their impact, and estimate optimization potential.

Detailed analysis Focusing and prioritizing the detailed analysis based on the outcome of the performance model. Gradually applying more advanced (and more costly) analysis techniques to understand the root causes of performance issues.

Reporting Recording the performance overview, analysis results and recommendations and reporting them. This allows for the key elements of the performance analysis to be accessed by other users or analysts, to be utilized for future analyses, and to prevent the loss of information.

In the following, we discuss the five steps in detail on the exemplary performance analysis of the CheSS library.

3.1 Measurement

Before the actual measurements we discussed the upcoming study with the customer to understand as good as possible his expectations and topics for the analysis. The main request for the analysis was to better understand the scalability of the application and to identify potential scalability limits. To this end, we agreed on two measurement sets recorded at MareNostrum 3, an Intel Sandy Bridge based system at BSC: The first set increases the number of MPI processes from 120, 240, 480 to 960. Thereby,

each node runs four MPI processes due to memory limitations. This measurement is used to study the scalability of the MPI communication and inter-node behavior. The second set studies the behavior depending on the ratio of MPI ranks to OpenMP threads. Each measurement uses 120 nodes with a ratio of MPI processes to threads of 960:1, 480:2, 240:4, and 120:8. Each node runs a total of eight processes/threads out of 16, again due to memory limitations. The measurements were performed by the user with the analyst's assistance using the above described configuration.

3.2 Overview and Focus of Analysis

Figure 1 shows timelines from the performance analyzer Paraver [4] of the execution using 120 MPI processes with one thread each. The timeline displays represent the behavior of an application along time and processes/threads and provide a general understanding of the application behavior and simple identification of phases and patterns. The timelines in Fig. 1 highlight different behavioral aspects evolving over time in a two-dimensional chart. The vertical axis includes the executing processes or threads; the horizontal axis represents the runtime behavior of the recorded application. The upper three screenshots depict the metric *Useful Duration*, i.e., time spent for computation outside of the parallel runtime (MPI or OpenMP); whereas the color gradient from green to blue represents the length of each compute phase from short to long, respectively; black marks time outside of useful computation, i.e., time in the parallel runtime. The bottom screenshots depicts the time spent in MPI communication.

The code executes three main phases *Init*, *Calc*, and *Last*. In the measured standalone version, *Init* makes up for about 80% of the runtime. However, this time is overrepresented in the standalone version; within a normal usage scenario, the *Calc* phase is dominating and, thus, the target of the analysis. The *Last* phase uses less than 0.1% of the time and can be neglected.

The *Calc* phase includes ten main phases of varying length. Thereby, the relative timing is identical for each phase, i.e. all phases are virtually the same but are differently stretched along time. We selected the fourth of these phases as focus of analysis (FOA).

The focus of analysis (bottom two screenshots in Fig. 1) itself consists of three compute phases: A first phase followed by calls to *MPI_Win_create*, *MPI_Get*, *MPI_Win_fence*, and *MPI_Win_free*. Second, the central compute phase followed by a large *MPI_Allreduce*, which is a global synchronization and collects the load imbalance of the previous phase. At the end, a short compute phase followed by another *MPI_Allreduce*.

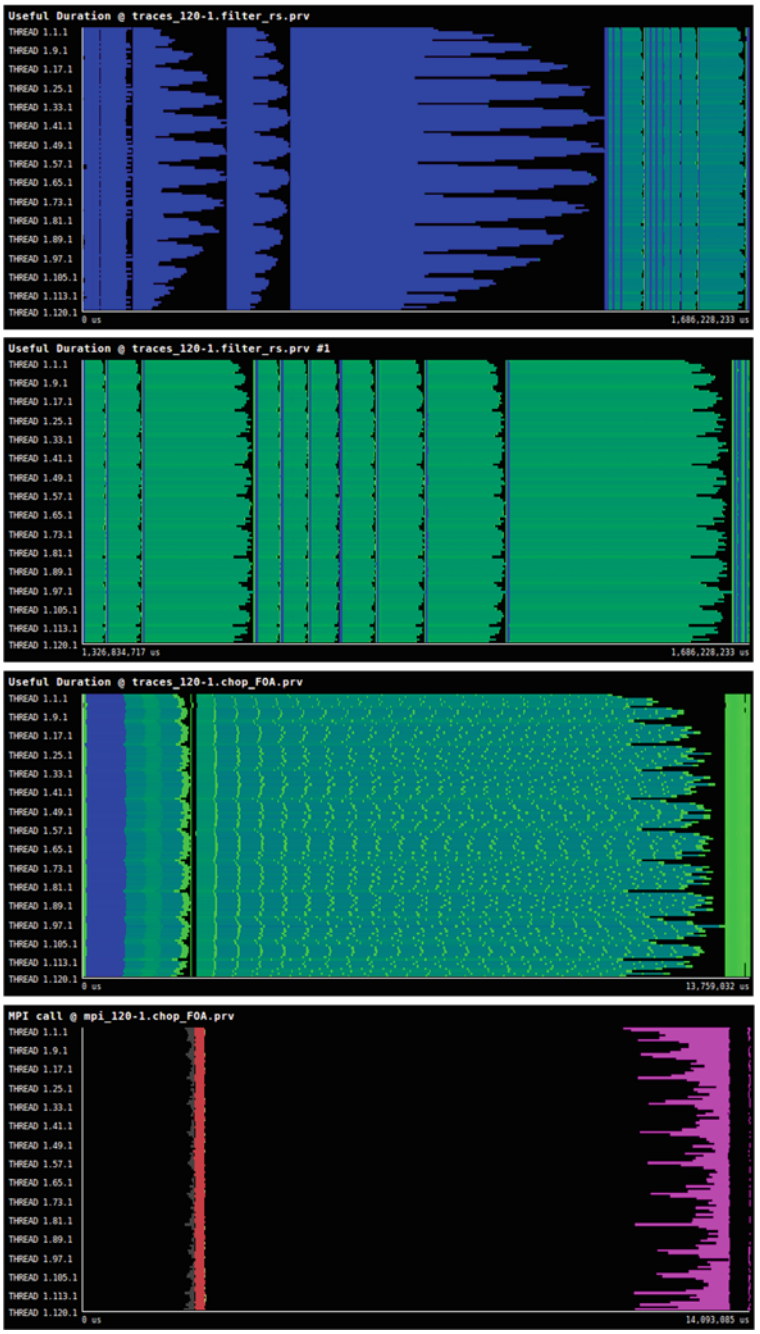


Fig. 1 Overview of the compute phases of the entire application (top), a zoom into the *calc* phase (second), a zoom into the fourth phase, which is the selected FOA (third), and the MPI communication for the FOA (bottom)

3.3 Performance Modeling

After determining the focus of analysis, we apply a performance model to the application phase selected as focus of analysis. The performance model combines fundamental performance factors that can be seen in Fig. 2. The performance model allows quantifying parallel efficiency and scalability with a single percentage value as well as providing an easy, high-level comparison of different executions.

The performance model computes the *global efficiency*, i.e. the overall performance rating, based on the two main components: *parallel efficiency* and *computation scalability*. The *parallel efficiency* provides an overall assessment of the parallel behavior of the application and is expressed as the product of *load balance* and *communication efficiency* [7]. *Load balance* reflects the efficiency loss due to imbalance in the total time spent in computation by each process. It is computed as the ratio between the average computing time and the maximum computation time of all processes. *Communication efficiency* describes the efficiency loss caused by communication between the processes and is computed as the ratio of the maximum time over all processes spent in computation and the complete runtime (computation time and communication time). It is also the product of *serialization efficiency* and *transfer efficiency*. *Serialization efficiency* measures the inefficiency due to dependencies in the execution leading to a serialization of the parallel computation. It is computed by simulating the execution with instantaneous (ideal) communication using Dimemas [8] and collecting the the maximum communication efficiency achieved by a single process. *Transfer efficiency* quantifies the performance loss caused by actual data transfers. It can be computed as the ratio between the communication efficiency of the real execution and the communication efficiency of an ideal execution (serialization efficiency). The parallel efficiency model is described in more detail in [7].

The *computation scalability* describes the evolution of the total time spent in computation of multiple executions and, therefore, is only meaningful for comparing multiple executions, e.g. with increasing core counts. The computation scalability of a given Recording the performance overview, analysis results and recommendations and reporting them. This allows for the key elements of the performance analysis

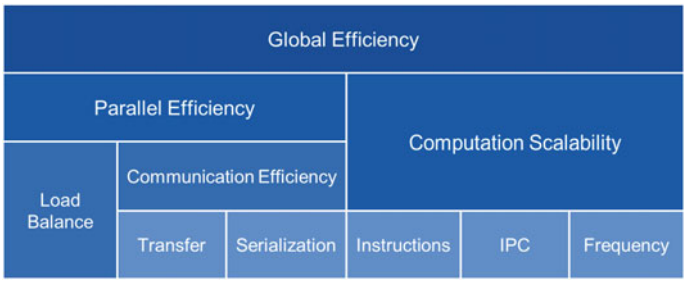


Fig. 2 Performance model: fundamental performance factors and efficiencies

to be accessed by other users or analysts, to be utilized for future analyses, and to prevent the loss of information.execution is computed as the ratio of the total computing time of a reference execution and the total computing time of the given execution. For instance, in measurement set with increasing core counts, the factor compares the total compute time of all executions to the smallest execution, which could be a serial or single-node execution. It can be further detailed in the scalability of the IPC (instructions per cycle), the instructions, and the frequency. *IPC scalability* reflects the evolution of the compute intensity over multiple measurements and is measured as the ratio of the total number of instruction and the total number of cycles spent in computation in comparison to the reference execution. *Instructions scalability* describes the evolution of the computational workload and is measured by the total number of instructions in computation over all processes in comparison to the reference execution. *Frequency scalability* measures the evolution of the clock frequency in the compute phases and is computed as the ratio between the total number of cycles and the total number of time spent in compute phases in comparison to the reference execution.

Table 1 shows the overview of the fundamental performance factors for the mpi-only measurements with 120, 240, 480, and 960 processes. While the performance model can be computed manually, Paraver's basic analysis package [2] computes all the performance factors automatically, which frees the user from manually collecting the data for the performance model and avoids potential errors in the process.

The observed global efficiency of the application decreases drastically from 90% with 120 processes to 52% with 960 processes. Table 1 reveals an efficiency loss in three main areas. First, the most severe issue is the decrease in computation scalability, which means the total amount of time spent in the computation phases is increasing. This is mainly due to the decreasing instructions scalability, i.e. with more processes the total amount instructions to solve the same problem is increased, which signals that a part of the total workload is replicated and, therefore, is not scalable. Second, an almost equally important issue is the decreasing communication

Table 1 Efficiency and scalability factors for the mpi-only executions with 120 to 960 processes

	120 (%)	240 (%)	480 (%)	960 (%)
Global efficiency	90.2	82.7	70.8	52.5
Parallel efficiency	90.2	85.9	78.9	68.5
→ Load balance	93.3	92.0	90.9	85.8
→ Comm efficiency	96.7	93.4	86.7	79.8
→ Serialization efficiency	99.8	99.8	99.8	99.8
→ Transfer efficiency	96.9	93.5	86.9	79.9
Computation scalability	100.0	96.3	89.7	79.0
→ IPC scalability	100.0	98.7	96.3	91.8
→ Instructions scalability	100.0	97.6	93.3	85.6
→ Frequency scalability	100.0	99.7	99.6	99.7

efficiency, which is almost entirely caused by a decreasing transfer efficiency. This indicates that relatively more time is spent in the actual data transfers for increasing core counts. The third issue is the decreasing load balance.

Breaking down the overall performance into distinct performance factors allows identifying the most important performance issues, quantifying their impact, and quantifying potential performance gains and can thus be used to guide the further, detailed analysis.

3.4 Detailed Analysis

The further analysis is focused and prioritized based on the outcome of the performance model. The detailed analysis investigates the revealed performance issues by gradually applying more advanced (and more costly) analysis techniques to understand the root causes of performance issues. In the following, we briefly discuss the detailed analysis, while an extended version can be found in the according analysis reports [9, 10]. For the analysis of the load balance and communication efficiency we use the mpi-only measurements, which allows to analysis parallel behavior at smaller scales than the hybrid MPI + OpenMP version (default).

Load Balance

Figure 3 depicts the effects of the load imbalance within the focus of analysis. While the left side gives an overview of the different duration of the compute phases, the right side shows a histogram with the distribution of compute phases based on their number of instructions. In the histogram each point represents a compute phase, whereas the vertical axis represents the processes and the horizontal axis the number of instructions performed in the given compute phase. The color gradient represents the duration of the compute phase and, thus, is identical with the timeline on the left. The further analysis reveals that, first, the load balance is directly related to the balance in instructions, i.e. the imbalance is defined by the workload distribution in the source code. Second, the nested d-shape of the load distribution hints to a decomposition where the most load is in the center of the domain and the center of each partition. In addition, the histogram allows to exactly pin-point the regions containing the load imbalance and link them to the according source code regions.

Communication Efficiency

Figure 4 compares the MPI communication with 120 (left) and 960 processes (right), whereas the time is scaled to show the entire focus of analysis in both cases.

The main contributor to the increasing communication time is the time spent in *MPI_Win_fence* (red), which increases from 26 ms with 120 processes to over 700 ms with 960 processes, This operation ensures that all previously initiated calls to *MPI_Get* are finished. Since the calls to *MPI_Get* are executed asynchronously,

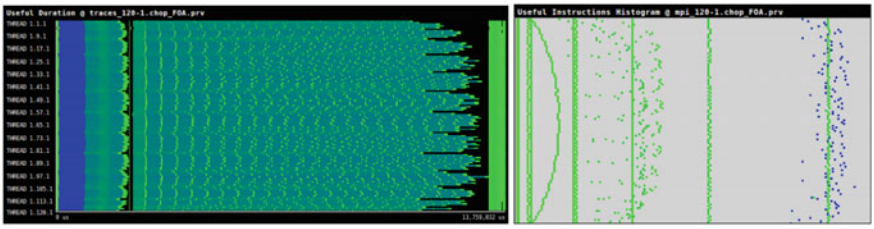


Fig. 3 Load balance: timeline with the compute phases (left) and a histogram showing the distribution of compute phases based on their number of instructions (right)

their time only shows the time needed to issue the operations. The actual execution of the transfer is represented in the time of *MPI_Win_fence*, i.e. the combined time is the real execution time of the *MPI_Get* operations. For each measurement, the number of *MPI_Get* operations per process is equal to the number of total MPI processes, e.g. 120, 240, 480, 960. Thus, the total number of *MPI_Get* operations increases quadratically with the number of processes. These *MPI_Get* operations are used to gather a distributed matrix, i.e. combined they mimic an *MPI_Allgatherv* with one-sided communication routines.

Computation Scalability

For the analysis of the computation scalability we use the second measurement set that contains the hybrid MPI + OpenMP measurements that keep the total amount of cores constant but vary the ratio of MPI processes to OpenMP threads (Fig. 5).

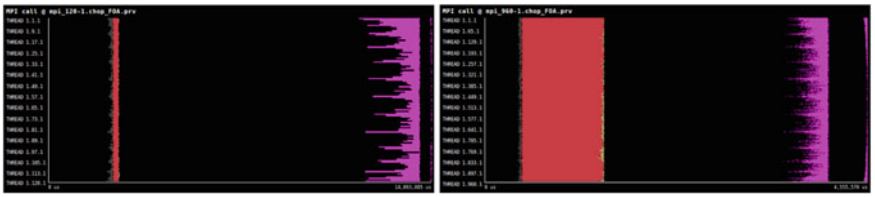


Fig. 4 Communication: comparing the communication with 120 (left) and 960 processes (right)

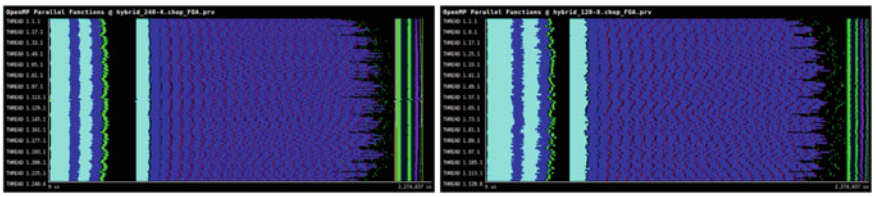


Fig. 5 OpenMP parallel regions with a 240:4 (left) and 120:8 (right) process:thread ratio

The hybrid version improves the parallel behavior drastically by increasing the load balance to 94.2% and the communication efficiency to 89.5%. In fact, the version with the most threads outperforms the other versions in almost every aspect leading to a speed up 2.0 over the mpi-only version. However, the computation scalability drops by over 5% from the 240:4 to the 120:8 measurements. The detailed analysis reveals two parallel loops (cyan, dark blue), where the compute efficiency (IPC) drops in the 120:8 version, while the workload (instructions) is identical. Using the clustering and tracking tools from the BSC tool suite [2] a variety of PAPI performance counters was compared between the two measurements, which identified stalls in the reservation station and reorder buffer as the main sources. Both effects are discussed in detail in the according performance report [10].

3.5 Reporting

The final step is documenting and reporting the analysis including the performance overview, analysis results and recommendations. While this step might seem dull and insignificant, it is essential to convey the results and recommendations to the user asking for the performance analysis. In addition, documenting the analysis allows for the key elements of the performance analysis to be accessed by other users or analysts, to be utilized for future analyses, and to prevent the loss of information.

4 Conclusions

In this paper we argue for a structured approach to performance analysis. The structured approach is motivated by the complexity of application behavior, the complexity of the performance issues and their sources and the resulting complexity of the performance analysis process itself. Identifying the main issues in the overwhelming amount of data and tool opportunities as well as quantifying their impact and potential for improvement can be tedious and time consuming at least, which leads to a frequent lack of quantified understanding of actual behavior and missing knowledge of the most productive direction of code refactoring.

We propose a structured approach to performance analysis cultivated and established for many years at BSC as well as the EU Centre of Excellence for Performance Optimization and Productivity. The structured approach consists of five steps from collecting a representative measurement set, to getting a general overview and defining the focus of analysis, identifying the main issues and areas for potential improvement with a statistical performance model, an in-depth analysis guided by the performance model, and, finally, a documenting and reporting the analyzed behavior, performance issues and recommendations. We discuss all steps along an exemplary performance analysis from the EU Centre of Excellence for Performance Optimization and Productivity and demonstrate their realization with the BSC tools.

Acknowledgements We gratefully acknowledge the support of the POP and MaX projects, which have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 676553 and 676598, respectively.

References

1. EU Centre of Excellence for Performance Optimization and Productivity (POP). http://pop-coe.eu/
2. BSC Tools. http://tools.bsc.es
3. Extrae instrumentation package. http://tools.bsc.es/extrae
4. Paraver: a flexible performance analysis tool. http://tools.bsc.es/paraver
5. Mohr, S., Dawson, W., Wagner, M., Caliste, D., Nakajima, T., Genovese, L.: Efficient computation of sparse matrix functions for large-scale electronic structure calculations: The cheSS library. J. Chem. Theory Comput. **13**(10), 4684–4698 (2017)
6. European Centre of Excellence Materials Design at the Exascale (MaX). http://www.max-centre.eu/
7. Rosas, C., Giménez, J., Labarta, J.: Scalability prediction for fundamental performance factors. Supercomput. Front. Innov. **1**(2), (2014)
8. Dimemas simulator. http://tools.bsc.es/dimemas
9. Wagner, M., Rosas, C., Giménez, J., Labarta, J.: CheSS/SIESTA Performance Assessment Report (POP_AR_32) (2016)
10. Wagner, M., Giménez, J., Labarta, J.: CheSS/SIESTA Performance Plan Report (POP_PP_11) (2017)

Counter Inspection Toolkit: Making Sense Out of Hardware Performance Events

Anthony Danalis, Heike Jagode, Hanumantharayappa, Sangamesh Ragate and Jack Dongarra

Abstract Hardware counters play an essential role in understanding the behavior of performance-critical applications, and inform any effort to identify opportunities for performance optimization. However, because modern hardware is becoming increasingly complex, the number of counters that are offered by the vendors increases and, in some cases, so does their complexity. In this paper we present a toolkit that aims to assist application developers invested in performance analysis by automatically categorizing and disambiguating performance counters. We present and discuss the set of microbenchmarks and analyses that we developed as part of our toolkit. We explain why they work and discuss the non-obvious reasons why some of our early benchmarks and analyses did not work in an effort to share with the rest of the community the wisdom we acquired from negative results.

1 Introduction

Improving application performance requires that the people undertaking the effort understand what the performance bottlenecks are. A key step in the process of understanding the factors that limit an application's performance is the examination of

A. Danalis (✉) · H. Jagode · J. Dongarra
Innovative Computing Laboratory, University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN 37996, USA
e-mail: adanalis@icl.utk.edu

H. Jagode
e-mail: jagode@icl.utk.edu

J. Dongarra
e-mail: dongarra@icl.utk.edu

Hanumantharayappa
Mathworks, 7700 Gleason drive, Apt 23D, Knoxville, TN 37919, USA
e-mail: Hanumanth.rpa@gmail.com

S. Ragate
Cerebras Systems, 428 Madera Ave, apt 5, Sunnyvale, CA 94086, USA
e-mail: sangamesh@cerebras.net

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_2

event counters that are recorded by the hardware during execution. Such counters can reveal the behavior of code segments with respect to the hardware. In particular, modern hardware contains performance monitoring units (PMUs), which count the events that take place:

- inside CPU cores, e.g., cache misses and branch related events,
- off-core, e.g., power consumption, and bytes read from memory controller,
- on completely separate hardware, such as network cards.

Many of these events have a rather straightforward meaning and can be mapped directly to the behavior of an application. However, when developers are interested in understanding the behavior of their code in great detail, the events that are counted inside CPU cores can prove to be quite challenging for two distinct reasons. First, the number of counters has been increasing over the years. Developers interested in how branches inside their code affect performance, or how good the cache locality of their memory access patterns is, would find themselves confronted with multiple events that relate to these concepts. Second, due to the increasing complexity of modern CPUs, many of the events that provide detailed information have complex descriptions and contain multiple flags that modify what exactly is being monitored. For example, when measuring requests that missed the Level 2 (L2) cache, on an Intel Haswell-EP CPU a developer can choose to count:

- Demand Data Read requests that miss the L2 cache.
- All demand requests that miss the L2 cache.
- Requests from the L2 hardware prefetchers that miss the L2 cache.
- Requests from the L1/L2/L3 hardware prefetchers or load software prefetches that miss the L2 cache.
- All requests that miss the L2 cache.

Clearly, relying on such short descriptions to choose the exact set of events and qulifiers needed to understand the performance bottlenecks of an application running on such complex hardware is not ideal.

Abstraction layers, such as the Performance Application Programming Interface (PAPI) [1], offer *derived* events that readily map to performance abstractions by offering combinations of actual *native* hardware events. However, such derived events hide details that could provide useful insights to the performance analyst.

In this paper, we present the Counter Inspection Toolkit (CIT), a collection of microbenchmarks and analyses aimed to automatically group hardware counters into logical groups based on what they are counting, and help performance-conscious application developers understand how counters relate to the behavior of code segments. We articulate the need for such a toolkit further by discussing how seemingly simple code segments can lead to non-obvious counter behavior, and discuss all the lessons learned, as well as our observations on details that can affect counter values and application performance.

In summary, this paper presents a body of work that aims to help developers who care about performance, but are not hardware wizards with a perfect understanding of chip design and counter semantics.

Fig. 1 Simple code segment

```
temp = 0;
do{
    temp++;
    if ( (temp % 2) == 0 ){
        global_var += 2;
    }
} while ( temp < size );
```

Fig. 2 Code segment with RNG

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( (result % 2) == 0 ){
        global_var += 2;
    }
} while ( temp < size );
```

2 Non-obvious Code Behavior

Let us consider the code shown in Fig. 1 and try to speculate on the number of conditional branches that will execute as a function of the parameter `size`.

We should first note that this code segment is oversimplified for demonstrative purposes, so we will assume that it has not been compiled with an aggressive optimization level, which would replace the whole loop with a simple expression due to the simplicity of the operations performed by the code. By examining the code, it is natural to infer that every iteration executes one conditional branch for the `if` statement and another for the termination condition of the `while` statement.

Examining the assembler code, shown in Fig. 3, annotated with the C code by gdb, helps enforce the assessment since the two condition branches ("`jne`" and "`jl`") can be seen in the code and no other branch instruction is present. The expectation can be verified experimentally by instrumenting the code with PAPI to count the number of conditional branch instructions that are executed by the loop, and, indeed, our experiments were in agreement with the theory.

Now, let us modify the previous program to include code that computes a random number. For simplicity, the random number generator (RNG) code is abstracted away in a macro that stores the random number in the variable `result` without making any calls to functions that would add branches to the execution. This variable is then used in the condition of the `if` statement as shown in Fig. 2.

Examining this new C code segment, one could infer that the number of conditional branches that will execute must remain the same, since the control flow of the code has not been affected by the modification. Furthermore, this assessment seems to be enforced by the corresponding assembler code, shown in Fig. 4, since the types and

```
52      do{
53          temp++;
   <310>:   mov    eax,DWORD PTR [...]
   <316>:   add    eax,0x1
   <319>:   mov    DWORD PTR [...],eax

54           if( (temp % 2) == 0){
   <325>:   mov    eax,DWORD PTR [...]
   <331>:   and    eax,0x1
   <334>:   test   eax,eax
   <336>:   jne    0x400e77 <353>

55                global_var += 2;
   <338>:   mov    eax,DWORD PTR [...]
   <344>:   add    eax,0x2
   <347>:   mov    DWORD PTR [...],eax

56          }
57      } while( temp < size );
   <353>:   mov    eax,DWORD PTR [...]
   <359>:   cmp    eax,DWORD PTR [...]
   <362>:   jl     0x400e4c <310>
```

Fig. 3 Disassembled simple code

relative positions of the conditional branch instructions in the new code segment are the same as before.

However, when we performed the same experiment as before we found the count of executed conditional branches to be equal to $2.5 \times$ `size`, which at first glance was surprising. Modifying the program further (by adding seemingly irrelevant work) offered an additional clue as to what is happening, although it initially seemed further perplexing.

In Fig. 5, we show an example where we modified the code shown in Fig. 2 by adding another call to the RNG after the branch (shown highlighted in red). Changing the code in this way makes the number of executed conditional branches go back to two per iteration.

Further clues can be found by counting the number of mispredicted branches, which turned out to be equal to $0.5 \times$ `size` in the examples shown in Figs. 2 and 5, which use the random number in the condition of the `if` statement, and zero[1] for the example shown in Fig. 1, which uses an easy to predict variable in the condition.

The final clue that helps explain the unintuitive discrepancy between these codes is the difference between the number of *retired* conditional branches and the number

[1] The actual count is not zero, but rather a small number due to noise caused by code not shown in the figures, such as the calls to `PAPI_start()` and `PAPI_stop()`. However, in our experiments this number did not grow when varying the variable `size`, so for large iteration counts the fraction of mispredicted branches approaches zero.

Fig. 4 Disassembled code with RNG

```
52      do{
53          temp++;
    <310>:  mov   eax,DWORD PTR [...]
    <316>:  add   eax,0x1
    <319>:  mov   DWORD PTR [...],eax

54          pseudo_random_generator();
    <325>:  mov   eax,DWORD PTR [...]
    ...
    <585>:  mov   DWORD PTR [...],eax

55          if ( ( result % 2 ) == 0 ){
    <591>:  mov   eax,DWORD PTR [...]
    <597>:  and   eax,0x1
    <600>:  test  eax,eax
    <602>:  jne   0x400f81 <619>

56              global_var += 2;
    <604>:  mov   eax,DWORD PTR [...]
    <610>:  add   eax,0x2
    <613>:  mov   DWORD PTR [...],eax

57          }
58      } while( temp < size );
    <619>:  mov   eax,DWORD PTR [...]
    <625>:  cmp   eax,DWORD PTR [...]
    <628>:  jl    0x400e4c <310>
```

Fig. 5 Code segment with RNG and redundant work

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( (result % 2) == 0 ){
        global_var += 2;
    }
    random_number( result );
} while( temp < size);
```

of *executed* conditional branches. Indeed, the number of retired conditional branches is always two per iteration in all three examples. Consulting the documentation of the hardware vendor [2], one can see that branch prediction leads to the instructions following the `if` branch to execute speculatively. In the cases where the speculation made a wrong prediction, the count of instructions that executed will include instructions that were canceled due to the misprediction. On the other hand, at-retirement counting only counts events that were committed to architectural state and ignores work that was performed speculatively and later discarded. In other words, as shown

in Figs. 3 and 4, since the branch due to the termination condition of the loop, "`jl`", is only a few instructions after the conditional branch of the `if` statement, which is predicted, the "`jl`" instruction will execute speculatively. In the code shown in Fig. 1, the speculation has no effect because the condition of the `if` statement is very regular and thus it is always predicted correctly. However, in the code shown in Fig. 2, the random variable will cause the condition to be mispredicted 50% of the time. Thus, in 50% of the iterations, the instructions that will execute speculatively (and among them the conditional branch "`jl`") will later have to be canceled—but they will be counted as executed nevertheless. This accounts for the extra $0.5 \times size$ factor in the count of the executed branches for this code (in comparison to the count of retired branches and in comparison to the executed branches for the code of Fig. 1.) To put it another way, the "`jne`" branch (of the `if` statement) is the one mispredicted, but the "`jl`" branch is the one with the 50% additional executions.

This explanation also covers the behavior of the code shown in Fig. 5. In this case the `jne` (`if`) branch is also mispredicted 50% of the time. However, the additional instructions between the `if` statement and the "`jl`" instruction push the latter instruction too far down the execution path and prevent the speculative execution from reaching it. I.e., the actual condition of the mispredicted branch, `jne`, is evaluated before the speculation can reach that far, and that whole path is discarded.

The set of microbenchmarks and analyses we are assembling together into the Counter Inspection Toolkit—which is the focus of this paper—aim to highlight such details in hardware counters and program execution, and identify connections between them.

3 Branch-Related Events

One of the principal goals of this work is to automatically categorize native events based on the higher-level concept they count. In other words, if two events have different counts for codes that stress different aspects of the architecture, then they belong in different groups; otherwise, they are grouped together. This endeavor is important because many native events that are exposed by hardware vendors support qualifiers that modify the actual hardware behavior that is being measured by the event. Furthermore, in some cases multiple qualifiers can be combined leading to a combinatorial explosion of possibilities. Therefore, we believe that application developers can benefit from a list which contains a short list of concepts that relate to branches, and the set of events and corresponding qualifiers that count each of these concepts.

For simplicity, in the rest of this paper, when we use the term "native event" we will refer to an event with qualifiers specified, not just the base event without qualifiers. In this section we will discuss the effort to categorize events that count branch-related execution.

Fig. 6 Code with direct branch

```
do{
    temp++;
    random_number( result );
    if( (result % 2) == 0 ){
        global_var += 1;
    }else {
        global_var += 2;
    }
} while( temp < size );
```

3.1 Design Choices

One of our design choices is to keep our microbenchmarks in C, instead of assembler. The reasoning is two-fold. Firstly, we want the benchmarks to be portable across architectures with incompatible instruction sets. Secondly, and perhaps more important, we want our benchmarks to be easy to read and comprehend by application developers even if they are not comfortable with assembler code. Meeting this design choice, however, can create challenges, since the compiler might try to rearrange code blocks to optimize execution, especially when optimization flags are used. As an example consider the code shown in Fig. 6.

Since the control flow of the `if-then-else` statement demands that the two blocks be mutually exclusive, one would expect that this code would be translated to assembler with one conditional branch and one direct branch (to choose one block and skip the other). This expectation turns out to be correct when no optimizations are performed during compilation. In Fig. 7, we show the assembler code which was generated when the "`-O0`" flag was passed to the compiler, and one can clearly identify the highlighted conditional and unconditional jumps (`jne` and `jmp`) used to implement the mutual exclusion of the two blocks (as well as the additional conditional jump, `jl`, for the loop termination condition).

However, if aggressive optimization flags are passed to the compiler, then the resulting assembler code does not contain a direct branch, as can be seen in Fig. 8. These kinds of discrepancies between what a developer assumes that the compiler will do and what the compiler actually does have made it challenging to uphold our design decision to write our microbenchmarks in C; but so far we have not found any insurmountable barrier. To address the specific problem discussed above, we wrote a microbenchmark (shown in Fig. 11f) that contains a `goto` statement and has a control flow graph that cannot be simplified by the compiler.

3.2 Controlling Branch Misprediction

The codes we showed in Figs. 2, 5, and 6 all contain `if` statements with conditions that compare the last bit of a random variable against zero. When a program does that, the expected rate of branch misprediction is 50%. Because of this, as we discussed

Fig. 7 Assembler with `-O0` flag

```
53      do{
    ...
56          if( (result % 2) == 0 ){
   <588>:   mov    0x200a5a(%rip),%eax
   <594>:   and    $0x1,%eax
   <597>:   test   %eax,%eax
   <599>:   jne    0x400e2e <618>

57              global_var += 1;
   <601>:   mov    0x200a21(%rip),%eax
   <607>:   add    $0x1,%eax
   <610>:   mov    %eax,0x200a18(%rip)
   <616>:   jmp    0x400e3d <633>

58          }else {
59              global_var += 2;
   <618>:   mov    0x200a10(%rip),%eax
   <624>:   add    $0x2,%eax
   <627>:   mov    %eax,0x200a07(%rip)

60          }
61      } while( temp < size );
   <633>:   mov    0x200a31(%rip),%eax
   <639>:   cmp    -0x14(%rbp),%eax
   <642>:   jl     0x400cf7 <307>
```

in Sect. 2, the count of executed conditional branches for the code in Fig. 2 was equal to $2 \times 0.5 + 3 \times 0.5 = 2.5$ per iteration. Going a step further, we can control the rate of branch misprediction by changing the condition to the one shown in Fig. 9. This allows us to control the rate of misprediction by assigning different values to the variable `K`.

Increasing the value of `K` leads to more branches evaluating to `false` than `true` (assuming a reasonable random number generator and an iteration count that is not trivially small). Therefore, we can expect the branch prediction unit to tend to predict `false` more often than `true`. A naive approximation would be to assume that as soon as $K > 2$ the branch prediction unit will always predict `false`. If that were the case then the count of executed conditional branches for the code in Fig. 9 would be equal to $2.0 \times \frac{K-1.0}{K} + 3.0 \times \frac{1}{K}$ per iteration, since only one out of `K` iterations would be mispredicted (and thus only one out of `K` iterations would speculatively execute an additional conditional jump). In Fig. 10 we plot this curve and the experimentally-measured count of executed conditional branches for this code.

As can be seen in the graph, when $K = 2$, the code degenerates to the original microbenchmark where the misprediction rate was 50% and thus the conditional branches that execute are 2.5. Also, as the value of `K` grows (and thus the condition

Fig. 8 Assembler with `-O3` flag

```
53      do{
    ...
56          if( (result % 2) == 0 ){
    <424>:  mov   0x200a82(%rip),%eax
    <430>:  test  $0x1,%al
    <432>:  je    0x400c08 <136>

57              global_var += 1;
    <136>:  mov   0x200b76(%rip),%eax
    <142>:  add   $0x1,%eax
    <145>:  mov   %eax,0x200b6d(%rip)

58          }else {
59              global_var += 2;
    <438>:  mov   0x200a48(%rip),%eax
    <444>:  add   $0x2,%eax
    <447>:  mov   %eax,0x200a3f(%rip)

60          }
61      } while( temp < size );
    <151>:  mov   0x200b97(%rip),%eax
    <157>:  cmp   %ebx,%eax
    <159>:  jge   0x400d53 <467>
    <453>:  mov   0x200a69(%rip),%eax
    <459>:  cmp   %ebx,%eax
    <461>:  jl    0x400c25 <165>
```

Fig. 9 Code with variable misprediction rate

```
temp = 0;
do{
    temp++;
    random_number( result );
    if ( (result % K) == 0 ){
        global_var += 2;
    }
} while( temp < size );
```

rarely evaluates to `true`), the measured value converges to the naive prediction, but for intermediate values the measured value is slightly above the curve. This suggests that the branch prediction unit tries to identify patterns in the random values instead of merely falling back to always predicting `false`, just because `false` is more frequent. Interestingly, this non-naive behavior of the branch prediction unit leads to more mispredictions than the naive approach of always predicting `false` would have led to.

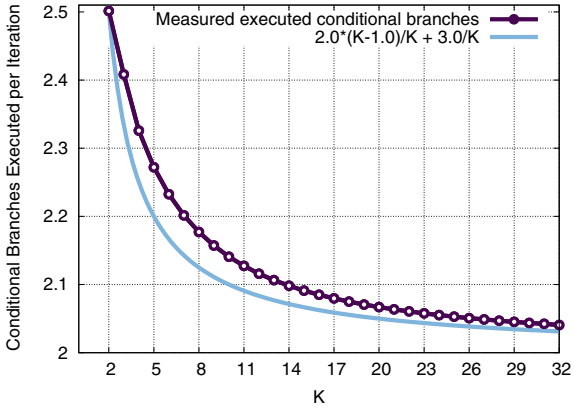


Fig. 10 Controling the misprediction rate

3.3 Event Categories

As we mentioned earlier, one of the goals of this work is to automatically categorize native events based on the hardware features they measure. The main categories for branch-related instructions are the following five:

CE: **Conditional Branches Executed**. This type of event counts the number of times a branch instruction that depends on a condition is executed by the hardware. Such instructions are generated from high-level language statements that affect the control flow, such as `IF`, or loops—where the conditional branch is used for the termination of the loop. Examples from the x86 instruction set are `je`/`jne` for "jump if (not) equal," `jge` for "jump if greater or equal," `jl` for "jump if lesser," and so on. Note that the instruction is counted as executed even if it executed speculatively, based on the misprediction of a previous branch (as discussed in Sect. 2). Also, a branch does not have to be taken to be considered executed. For example, the branch `if( 0 == 1 )` will never be taken but the hardware will still execute it to decide not to take it (assuming the compiler did not optimize it away).

CR: **Conditional Branches Retired**. This type of event involves the same instructions as the event above (i.e., `je`, `jne`, `jge`, `jl`, etc), but in this case the execution of a branch has to be committed to architectural state for it to be counted as retired. This means that either (a) the execution of the branch was not part of speculative execution, or (b) the execution of the branch was part of speculative execution and the speculation was proven correct. Section 2 contains an extensive discussion of the difference between *executed* and *retired* instructions.

T: **Conditional Branches Taken**. This type of event counts only the branches where the condition evaluated to `true` and the branch was actually taken. For this type of event, the differentiation between `executed` and `retired` is

microarchitecture specific, as not all CPUs, even from the same vendor, offer both versions.

D: **Direct Branches Executed**. This type of event counts the number of times an unconditional branch instruction is executed by the hardware. Such instructions are commonly generated from compilers to support the control flow of `if-then-else` statements, or to translate high-level language statements such as `goto`.

M: **Branches Mispredicted**. This type of event counts the number of times the branch prediction hardware made a wrong prediction. For example, if it were to predict that a branch will be taken because it predicts that the condition will evaluate to `true`, but it is not taken because the condition turns out to be `false`.

In order to categorize the native events of an architecture in these five groups, we wrote a set of microbenchmarks that all contain branch instructions, some of which are shown in Fig. 11. As can be seen from the figure, the benchmarks are similar in many ways, but they also diverge from one another. As a result, an event that falls in any of the five categories (CE, CR, T, D, M) should give a set of values (when measured for the different benchmarks) that is not the same across all benchmarks. Furthermore, depending on the category of the event, the set of values for the different benchmarks will be different. Therefore, using the output of these benchmarks one could categorize events into classes.

In the following section we discuss the approaches we took to automate the classification of different events based on these benchmarks, as well as the lessons we learned.

3.4 Analysis of Benchmark Results

All the analysis techniques we have tried so far rely on the same basic methodology. Specifically, for each event that we are trying to classify we run each benchmark multiple times, varying the iteration count (which is controlled by the variable `size`) so that we get a curve from each benchmark for each event.

Our first attempt to associate events with categories was by using the Pearson correlation coefficient [3]. The idea was that a given benchmark stresses a particular category of events, therefore each benchmark would produce a growing curve for the events that belong to this category and a flat curve for all others. Let us take the code shown in Fig. 11a as an example, and let us call it bench1, for simplicity. Now, consider that for every possible native event on a system, we make multiple runs of bench1, every time setting a different value to the variable `size`. This benchmark is expected to only trigger events that measure conditional branches (CE and CR) and taken branches (T). Therefore, in the data sets resulting from these runs we should witness a correlation between the control variable `size` and the measured variable only for events that measure conditional branches (CE and CR) and taken branches

```
do{

    if ( temp < (size/2) ){
        global_var2 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(a): {2, 2, 1.5, 0, 0}

```
do{
    global_var2 += 2;
    if ( temp < global_var2 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(b): {2, 2, 1, 0, 0}

```
do{

    global_var2 += 2;
    if ( temp > global_var2 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(c): {2, 2, 2, 0, 0}

```
do{
    random_number( result );
    global_var2 += 2;
    if ( (result % 2) == 0 ){
        global_var1 += 2;
    }
    random_number( result );
    temp++;
}while( temp < size );
```

(d): {2, 2, 1.5, 0, 0.5}

```
do{
    random_number( result );
    global_var2 += 2;
    if ( (result % 2) == 0 ){
        global_var1 += 2;
    }

    temp++;

}while( temp < size );
```

(e): {2.5, 2, 1.5, 0, 0.5}

```
do{
    global_var2 += 2;
    if ( temp < global_var2 ){
        global_var1 += 2;
        goto zz;
    }
    random_number( result );
zz: temp++;
    random_number( result );
}while( temp < size );
```

(f): {2, 2, 1, 1, 0}

```
do{
    global_var2 += 2;
    temp++;
}while( temp < size );
```

(g): {1, 1, 1, 0, 0}

Fig. 11 Benchmark kernels and their expected values for the branch event categories (CE, CR, T, D, M)

(T). While the correlation coefficient does distinguish the relevant events from the majority of the irrelevant ones, it proved to be a very crude tool unfit for automatic categorization. There are two reasons for this failure. Firstly, we witnessed a few false positives due to noise, and multiple false positives due to events that are completely unrelated to branches (i.e., number of executed instructions), but are legitimately correlated with the iteration count. Second, this technique has a fundamental flaw in

that most of our microbenchmarks trigger events from more than one branch category at the same time, and this technique is unable to differentiate between them.

A better solution is to use the data from each benchmark to calculate a slope for each event using *least squares fitting*. Each of our benchmarks is expected to trigger a known number of events in each category, per iteration, as shown in the captions in Fig. 11. Taking again bench1 as an example and running it multiple times (while varying `size`) for each native event will generate a unique data set for that event. Consider now that we have such a data set by running bench1 and measuring event E_i (e.g., "`BR_INST_EXEC:TAKEN_COND`"). Fitting this data set using least squares will generate a measured slope, β_m. Then this slope can be compared with the expected slope, β_e, for each event category—so for bench1 we would compare β_m against the values 2, 2 and 1.5, as shown in the caption of Fig. 11a. If β_m matches one of these values, then the event E_i belongs to the corresponding category. Using the example of "`BR_INST_EXEC:TAKEN_COND`," we expect the measured slope to match the value 1.5, which reveals that this event belongs to the category "**T** (conditional branches taken)."

4 Cache-Related Events

The degree to which a code is reusing the caches of a CPU usually has a substantial effect on performance. To help assess how well cache reuse is achieved by an application, hardware vendors offer multiple events that count different behaviors of the cache hierarchy. Unfortunately, the complexity of modern cache subsystems has led to multiple such events, sometimes with non-obvious names and functionalities.

To assist developers in choosing which event to use, and understanding what each event measures, we used microbenchmarks that stress the cache subsystem. The key idea underlying our codes is to control the way memory is accessed, as well as the amount of memory that is accessed, and observe how the measured events change.

We use a technique known as pointer chaining (or pointer chasing), which is common in the benchmark literature [4–6]. The basic idea is to use an array of integers, each long enough to hold a pointer (`uint64_t`). Then, each element of the array is made to point to another element of the array following a random pattern. This creates a "pointer chain." After this setup phase, the program can start a "pointer chase" where the first element of the array is accessed and the value it contains becomes the next element to be accessed, and so on.

The setup of the array can happen off-line, so even an expensive pseudo-random number generator (RNG) can be used, such as the function `random()`—commonly found in POSIX and BSD systems—which employs a non-linear, additive feedback and has a period of $\approx 16 \cdot (2^{31} - 1)$. Using such an RNG, the generated pattern becomes exceedingly difficult for the prefetching hardware to guess.

Figures 12 and 13 show the results of running our memory access benchmark with a variable array size while measuring the value of different events. As can be seen, the values of the different events show sharp transitions from 0 to 100% at

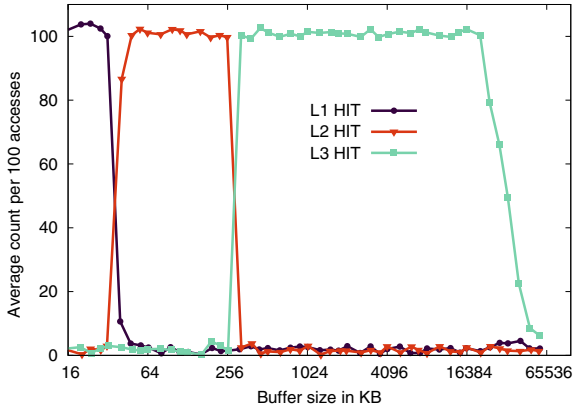


Fig. 12 Cache HIT related events

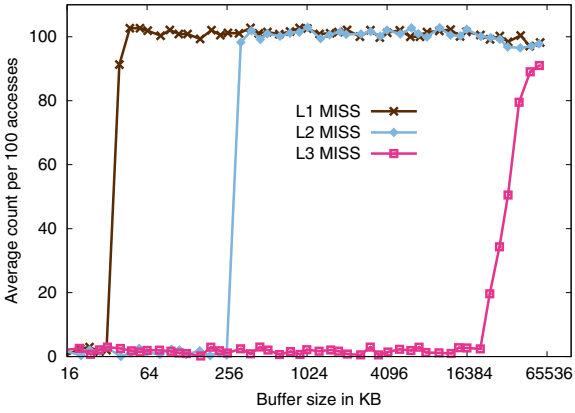


Fig. 13 Cache MISS related events

the boundaries of the different caches—in this example, L1 = 32 KB, L2 = 256 KB and L3 = 32MB. These sharp transitions can function as signatures that enable us to categorize events based on whether they measure a *hit* or a *miss*, and which cache level they relate to.

Another interesting behavior that we can probe with our microbenchmarks is prefetching. One can assume that when a miss occurs, the cache fetches more consecutive lines than the one requested speculatively, expecting spatial locality in future memory accesses. If this is the case, then altering the minimum distance between memory accesses (i.e., changing the size of our minimum accessible unit) should result in a different hit rate. In Fig. 14 we show three curves that correspond to three different minimum unit sizes on an architecture where the actual L2 line size is 64B. Indeed, even when the buffer size exceeds the size of the L2 (256 KB), for small unit sizes the hit rate remains surprisingly high. However, when the unit size increases, which means that the additional consecutive lines are never accessed, the hit rate sharply drops to a very small value when the buffer size exceeds the size of the L2 cache.

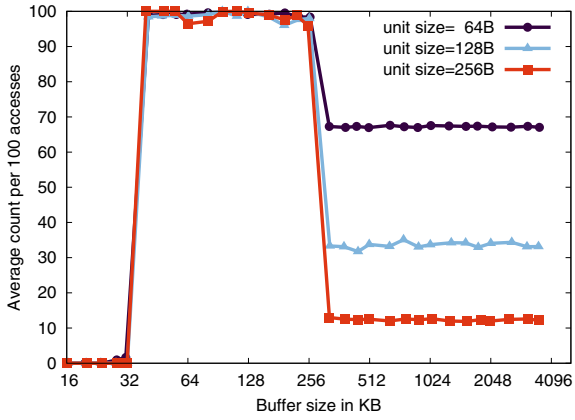


Fig. 14 Effect of unit size on L2 hit ratio

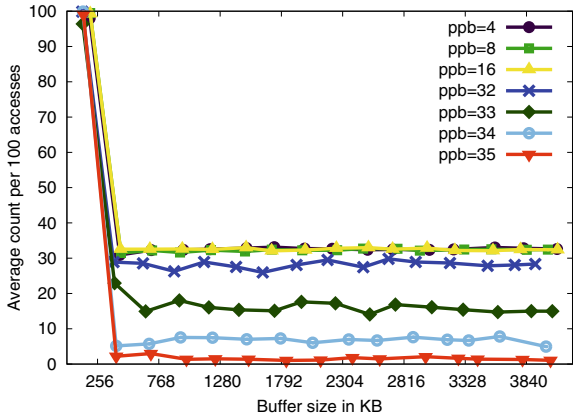


Fig. 15 Effect of block size on L2 hit ratio

Figure 15 shows a different experiment that also tests aspects of prefetching. Here, we kept the minimum unit size constant (128B) across runs, but we altered the way we created the pointer chains. Namely, regardless of the buffer size, the buffer is segmented into logical blocks and each block has its own chain. Therefore, all the elements in the first block are accessed first, then all the elements in the second block, and so on. The rationale behind this design is to restrict the number of operating system pages in each block so that the number of translation lookaside buffer (TLB) misses is minimized during pointer chasing (since TLB misses are often more expensive than cache misses, and pollute the L3 cache, and thus affect the behavior of the memory hierarchy). As the size of the logical blocks grows, pressure on the cache prefetcher increases, since more and more pages need to be monitored. Indeed, as can be seen in Fig. 15, when the number of pages per block (ppb) is 16 or less, the hit rate remains high even past the size of the L2 cache ($\approx$30%). However, as the number of pages per block grows beyond 32, even for values barely

above 32, the cache hit rate drops quickly, all the way to almost zero. This behavior can then be used to categorize events that relate to cache prefetching.

4.1 Assisting Developers with Code Optimization

As we mentioned earlier, one of the main driving forces behind this work is to assist performance-conscious application developers in understanding the behavior of the hardware so they can optimize their codes. As a result, we are interested in behaviors that are demonstrated by microbenchmarks and can be used to make design decisions in larger applications.

Consider a system with an L1 cache of size 32 KB and an application that accesses a buffer of size 32 KB (or less), such that smaller blocks, e.g., 4 KB, are accessed one after the other sequentially—instead of the application accessing elements spread out in the whole 32 KB buffer. Consider also that after the code accesses the last block, it goes back to the beginning of the buffer and accesses all the blocks again, and this loop continues for many iterations. This behavior is shown schematically in the first line of Fig. 16. Since the buffer fully fits in the L1 cache, there will be good cache reuse and therefore low average memory access time. This case is shown graphically in the first line of Fig. 16. However, if we grow the buffer to $32 + 4 = 36$ KB something interesting happens. When the application now accesses the last block, which does not fit in the L1 cache along with all previous ones, 4 KB from the previously accessed data has to be evicted. Since the very first block was the least recently used (LRU) data, and since LRU is a popular replacement policy, the cache will evict the whole first 4 KB block. As a consequence, when the code comes around to access the first block again, that block will not be in the L1 cache. Even worse, these new accesses to the first block will evict the second block, which is now the least recently used one. As the code continues, each block will evict the

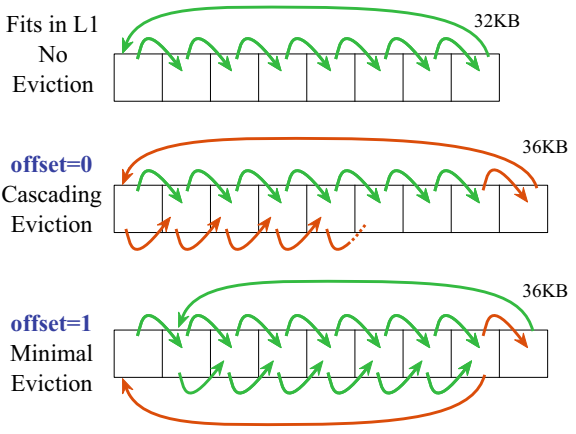


Fig. 16 Access patterns with and without offset

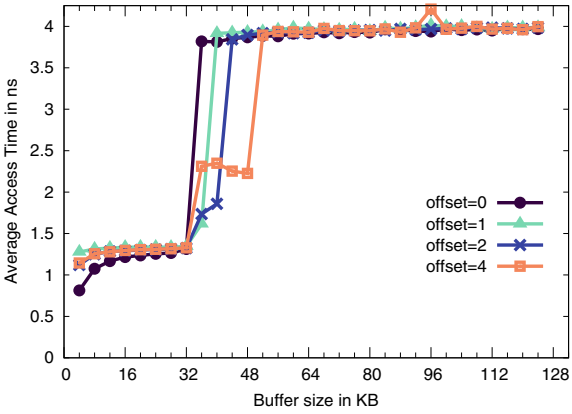


Fig. 17 Cache replacement and access pattern

next, and every memory access will lead to a miss in the L1 cache, so it will be served at the latency of the L2 cache. This behavior is shown in the second line of Fig. 16.

However, if the application was written by someone aware of this behavior, much better locality could have been achieved. Specifically, if after the last 4 KB block is accessed the code skips the first block and accesses all others, leaving the first block for last, then this cascading series of evictions would be interrupted and most of the blocks would be served from the L1, leading to a much lower average access time, as shown in the last line of Fig. 16. Hereafter, we will use the term *offset* = 1 to describe this approach. Growing the buffer by another 4 KB block would nullify the benefits of this technique, but using *offset* = 2 would still work, as would larger offsets for larger buffers. The efficiency of this technique is demonstrated in the performance graph shown in Fig. 17, where we show the result of using these access patterns on a machine with L1 latency $\approx$ 1 ns and L2 latency $\approx$ 4 ns.

5 Categorizing Events Automatically

As we discussed in Sect. 3.4 the count of branch events grows linearly as we increase the iteration count of our benchmarks. Cache events, however, tend to follow step functions that jump abruptly between extreme values. Nevertheless, there are ways to automatically categorize events from both groups: by turning the information we generate through our benchmarks into signatures. In Table 1 we show the expected values for the branch event categories we described in Sect. 3.3 for all the benchmarks we showed in Fig. 11. As it is easy to see from the table, no two rows are identical. Therefore, if for every event that we test we use the results of all benchmarks together, then we obtain a signature that is unique for each event category.

To make the concept of the signature clearer, consider as an example that we perform a test where we run these seven benchmarks, and in every run we measure

Table 1 Expected values for different branch event categories across multiple benchmarks

	Bench1	Bench2	Bench3	Bench4	Bench5	Bench6	Bench7
CE	2	2	2	2	2.5	2	1
CR	2	2	2	2	2	2	1
T	1.5	1	2	1.5	1.5	1	1
D	0	0	0	0	0	1	0
M	0	0	0	0.5	0.5	0	0

the native event "`BR_INST_EXEC:ALL_COND`." We run each benchmark multiple times, and in every run we vary the iteration count. Subsequently, we process the measurements taken from each benchmark to obtain a slope. If the slope values we obtain from the different benchmarks are "2, 2, 2, 2, 2.5, 2, 1" then by consulting this table we can uniquely identify this native event as belonging to the category "**CE**" (conditional branches executed).

In reality, our measurement will contain noise, so the values acquired from measurements are unlikely to exactly match the values in this table. To address the noise and suppress irrelevant native events the measurements of which happen to have a slope similar to an expected one, we also incorporated the correlation coefficient (r^2) of the fitting into our "slope goodness function," which is presented below.

$$goodness = \mathrm{e}^{-2\cdot(\beta_m \cdot r^2 - \beta_e)^2} \tag{1}$$

We chose this formula because it has the shape of the normal curve, which is forgiving for small variations, but quickly becomes punishing for larger ones. As a result, when the measured slope β_m is close to the expected slope β_e and the correlation coefficient r^2 is very close to one (indicating a good fit), then this formula will produce a number very close to one. However, if the measured slope diverges from the expected slope, or if the correlation coefficient is low, then the formula will produce a number close to zero.[2]

For each native event E_i the seven different benchmarks will produce seven different measured slopes, $\beta_m^1, \beta_m^2 \cdots \beta_m^7$. Using this formula we can compare these slopes against the different rows of Table 1 and get a quantitative assessment of the proximity of event E_i to the category represented by each row.

As we mentioned earlier, the benchmarks that stress cache-related events do not produce slopes, but rather step functions. However, these step functions can be readily converted to signatures if we ignore the multiple values at each plateau and we only keep the actual transitions, which as we showed in Figs. 12 and 13 are unique for each cache event category.

[2]Other, more sophisticated goodness functions, such as Pearson's χ^2 test [7], could be used to assist in the analysis of the measurements, but in our experiments we found that the simple formula in Eq. 1 is sufficient.

As part of our research effort, we also developed a long short-term memory (LSTM) neural network that we trained to recognize the patterns produced by our benchmarks. It proved to by fairly successful when we tried it on architectures other than the one we trained it on (i.e., we trained it on Intel x86 and used it on IBM Power8), but the details of this approach are outside the scope of this paper.

Using our benchmarks along with the analysis described in this section produces an automatic categorization of events in a system where the user does not already know which native events belong to each category. Conversely, in a system where the user knows what each native event is supposed to measure, our automatic event categorization can be used for verification of specific native events.

6 Related Work

There are several tools and APIs for accessing hardware counters. PAPI [1] is one of the most widespread, due in part to its strength as a cross-platform and cross-architecture API. PAPI provides short descriptions of the events that can be measured, but these descriptions are not always self-explanatory, especially so for application developers who are not experts on a given architecture. For Linux platforms the *perf tool* [8] makes use of the *perf_event* API, which is part of the Linux kernel. perf_event is even more low-level and the information returned requires considerable interpretation to be useful to application developers.

Further, processor vendors supply tools for reading performance counter results, such as *Intel VTune* [9], *Intel VTune Amplifier*, *Intel PTU* [10], and *AMD's CodeAnalyst* [11], but none of these tools and APIs comes with a set of benchmarks whose behavior is easy to understand and yet demonstrates behaviors of the underlying hardware that affect application performance.

The closest to the work presented in this paper is the *likwid* lightweight performance tools project [12]. In addition to enabling the user to access performance counters through direct access to the hardware, likwid offers a set of microbenchmarks that stress different aspects of the hardware. However, unlike our work, these microbenchmarks are written in a custom low-level language that maps directly to x86 assembler and are aimed at calibrating the tool—not to educate application developers about the higher-level meaning of different events, or to help them discover the meaning of events on diverse architectures.

In terms of system benchmarks, there are multiple projects [5, 6, 13–19] aiming to achieve different goals, such as analyzing the micro-architecture of a specific platform in great detail, or offering an extendable base of micro-kernels so that more complex benchmarks can be built on top of them. While we have learned valuable lessons from several of these efforts, and we have borrowed techniques such as the pointer chaining, none of these benchmarks was developed having in mind the goals of characterizing hardware events to provide application developers with a more intuitive high-level understanding of the concepts that are being counted.

7 Conclusions

In this paper we presented our work on the Counter Inspection Toolkit, a collection of microbenchmarks and analyses developed in an effort to categorize and abstract hardware events and map them to higher-level performance concepts. The driving force behind this effort has been our desire to illuminate the way for application developers who are keen on performance optimization, but are not experts on every esoteric detail of the latest hardware micro-architecture.

We discussed several interesting and non-obvious findings, we demonstrated the feasibility of categorizing events into logical groups, and discussed how automatic analyses can be employed to assist in this categorization. In future work, we are planning to extend the toolkit by adding multithreaded benchmarks that stress parts of the hardware related to resource sharing. We also plan to publicly release the code.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1450429.

References

1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
2. Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 (2017)
3. Pearson, K.: Notes on regression and inheritance in the case of two parents. Proc. R. Soc. Lond. **58**, 240–242 (1895)
4. Danalis, A., Luszczek, P., Marin, G., Vetter, J.S., Dongarra, J.: Blackjackbench: portable hardware characterization with automated results analysis. Comput. J. **57**(7), 1002 (2014)
5. McVoy, L., Staelin, C.: `lmbench`: Portable tools for performance analysis. In: Proceedings of the Annual Technical Conference on USENIX 1996 Annual Technical Conference ATEC'96, pp. 23–23. USENIX Association, Berkeley, CA, USA, 24–26 Jan 1996
6. Mucci, P.J., London, K.: The CacheBench Report. Technical report, Computer Science Department, University of Tennessee, Knoxville, TN (1998)
7. Pearson, K.: On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. Philos. Mag. **5**(50), 157–175 (1900)
8. Molnar, I.: perf: Linux profiling with performance counters (2009). https://perf.wiki.kernel.org/
9. Wolf III, J.H..: Programming Methods for the Pentium III Processor's Streaming SIMD Extensions Using the VTune™ Performance Enhancement Environment. Intel Corporation (1999)
10. Intel Performance Tuning Utility. http://software.intel.com/en-us/articles/intel-performance-tuning-utility/
11. Drongowski, P.J.: An introduction to analysis and optimization with AMD Code Analyst™ Performance Analyzer. Advanced Micro Devices, Inc. (2008)
12. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of the First International Workshop on Parallel Software Tools and Tool Infrastructures, September 2010

13. Dongarra, J., Moore, S., Mucci, P., Seymour, K., You, H.: Accurate cache and TLB characterization using hardware counters. In: Marian Bubak, G., van Albada, D., Sloot, P.M.A., Dongarra, J. (eds.) International Conference on Computational Science, volume 3036 of Lecture Notes in Computer Science, pp. III:432–439. Krakow Poland, June 2004. Springer, Heidelberg. ISBN 3-540-22114-X
14. Duchateau, A.X., Sidelnik, A., Garzarán, M.J., Padua, D.A.: P-ray: a suite of micro-benchmarks for multi-core architectures. In: Proceeding of the 21st International Workshop on Languages and Compilers for Parallel Computing (LCPC'08)
15. Gonzalez-Dominguez, J., Taboada, G.L., Fraguela, B.B., Martin, M.J., Tourio, J.: Servet: a benchmark suite for autotuning on multicore clusters. In: IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–10. IEEE Computer Society, Atlanta, GA, 19–23 Apr 2010. https://doi.org/10.1109/IPDPS.2010.5470358
16. Molka, D., Hackenberg, D., Schone, R., Muller, M.S.: Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques PACT '09, pp. 261–270, Raleigh, North Carolina, September 12–16. IEEE Computer Society, DC, USA, Washington (2009)
17. Staelin, C., McVoy, L.: mhz: Anatomy of a micro-benchmark. In: USENIX 1998 Annual Technical Conference, pp. 155–166. USENIX Association, New Orleans, Louisiana, 15–18 Jan 1998
18. Yotov, K., Jackson, S., Steele, T., Pingali, K., Stodghill, P.: Automatic measurement of instruction cache capacity. In: Proceedings of the 18th Workshop on Languages and Compilers for Parallel Computing (LCPC), pp. 230–243. Springer, Hawthorne, New York, 20–22 Oct 2005
19. Yotov, K., Pingali, K., Stodghill, P.: Automatic measurement of memory hierarchy parameters. SIGMETRICS Perform. Eval. Rev. **33**(1), 181–192 (2005)

ASSIST: An FDO Source-to-Source Transformation Tool for HPC Applications

Youenn Lebras, Andres S. Charif Rubial, Romain Dolbeau and William Jalby

Abstract The complexity and the diversity of computer architectures have dramaticaly evolved over the last decade, which makes it impossible to manually optimize codes for all these architectures. In addition, compilers must remain conservative with respect to their optimization choices because of their static cost model. One way to guide them is to use feedback data from data profiling of a representative training dataset (FDO/PGO) for a given application. It then becomes possible, based on that knowledge, to add specific compiler directives and/or flags to enhance performance. Moreover, automatic transformations simplifying portions of the application (e.g. specialization) can be applied. In this paper we present ASSIST, a directive-oriented source-to-source manipulation tool that aims at providing such assistance. The tool is integrated into the MAQAO toolset and takes advantage of all the available static and dynamic profiling data produced by the other tools. It also features a set of code transformations triggered by directives. The combination of both leads to an autotuning process that helps users to keep their code as generic as possible whilst also benefiting from a performance gain related to feedback or user knowledge. We demonstrate how we can build a compiler's PGO-like tool and compare our first results to the Intel compiler PGO mode.

Y. Lebras · A. S. Charif Rubial · W. Jalby (✉)
UVSQ/Exascale Computing Research, 45 avenue des Etats Unis,
78000 Versailles, France
e-mail: william.jalby@uvsq.fr

A. S. Charif Rubial
PeXL, Versailles, France

Y. Lebras · A. S. Charif Rubial · W. Jalby
Exascale Research Computing, Versailles, France

R. Dolbeau
Atos, Bezons, France

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_3

1 Introduction

The new generation of high performance cores/processors is heavily relying on increased vector length and advanced memory hierarchies to deliver higher performance levels. Such trends stresses the importance of data access optimizations and vectorization. The compiler is the first classical approach to address these issues. Unfortunately they suffer from two major limitations: first their search for code transformations is de facto limited because searching through a huge space of transformations without specific "tips" is extremely expensive and second because the choice of the applied transformation is entirely based on static information, running the risk of missing the right target.

The code developer with his knowledge of the code which might be augmented by the use of performance tools to characterize code behavior can help the compiler in identifying the right transformations. He can annotate source code either through custom directives [5, 31, 35], comments [19, 36] or using Domain specific Language (DSLs) [10, 12, 13, 27, 33]. Directives are simple but less powerful when compared to DSLs which can handle very advanced patterns at the price of complexity. From the point of view of a a regular application developer, directives provide the best compromise (expressiveness v.s. complexity). However, the resulting source code may end up bloated by optimization transformations (e.g. tiling), special cases or even useless modifications. It is even worse if users need to target multiple architectures (e.g. x86 and GPU or ARM). Finally, all of these source code edits are put on developers shoulders, impacting its productivity, creating the risk of inserting useless or detrimental annotations or much worse, introducing bugs.

A very promising approach to relieve the user from these tedious edits is to use feedback data optimization (FDO). Feedback data is any kind of data that can be gathered on a code and can be used to characterize it from a performance perspective. It should be noted that in the literature FDO and Profile Guided Optimizations (PGO) have the same meaning. We will use FDO in the rest of this article because, in our opinion, it is more generic. For instance feedback data could be a small trace which differs from a profile. One example of FDO is the FDO modes embedded with production compilers (Intel, GCC and more recently LLVM) known as *pgo* [24] and *autofdo* [9, 20]. A typical FDO process encompasses three steps: producing an instrumented binary using a special compiler flag(s); executing the resulting binary to obtain a profile; and finally, using feedback data during the compilation process to produce a new version that is supposed to be more efficient. However, in the current FDO implementations the level of information gathered at run time is limited and second the transformation space searched is also limited. Both limitations have a strong detrimental impact on the efficiency of the transformations applied. We will demonstrate that by being more aggressive on information gathering combining static and dynamic information, and on code transformations, substantial performance gains can be obtained.

This paper presents ASSIST, a directive-oriented source-to-source manipulation tool. It is able to guide code transformations based on static and dynamic feedback. It aims at providing assistance with respect to productivity and performance efficiency. The main contributions of our tool are to provide: a new open source FDO tool using both static and dynamic feedback while existing ones only use dynamic feedback; a more flexible alternative to compilers PGO/AutoFDO modes while being complementary; elaborated transformations such as loop and function specialization including our block vectorization transformation which helps the compiler to harness vectorization.

This paper is organized as follows: Section 2 provides an overview of our approach. Then Sect. 3 describes the design and implementation of the tool. The following Sect. 4 presents the available transformations. In Sect. 5 we will study the experimental results. Related work is listed in Sect. 6 before concluding and mentioning future work in Sect. 7.

2 Background and Goals

The MAQAO toolset [3] focuses on the performance evaluation and optimization of binary applications. The toolset features multiple tools [7, 8, 22, 29] which share the same rationale, namely pinpointing issues at source level and providing users with hints and even workarounds to be applied. In order to efficiently use these tools, a methodology [4] has been proposed. It aims at providing a way to filter all the data collected from the performance evaluation tools and classify them according to their return on investment (ROI) metrics.

Working at binary level has the advantage of evaluating the code that will really be executed (i.e. after compiler modifications). However, the main drawback is that we do not have access to the source code. A match between assembly level and high level source structures like functions/loops has to be based on debug information provided by the compiler. According to the optimization level, debug information is more or less accurate. This is due to the transformations/optimizations (e.g. inlining) performed by the compiler. It is also impossible to control all code properties that could help to provide more accurate results when combined with binary analyses. Enabling MAQAO to deal with source code would allow more accurate analyses. MAQAO can pinpoint different kinds of performance issues (i.e. diagnosis). The next step is to try to fix them at source level.

When performing optimizations on real applications we face three main concerns: selecting which transformations to apply to fix issues; minimizing code bloating due to transformations like hand-coded (function/loop) specialization; avoiding having to apply tedious (when not error-prone) transformations. The main goal of our approach is to help users increase performance without reducing the programming productivity.

3 Design and Implementation

In order to achieve the goals listed in the previous section, ASSIST must handle source code manipulation and harness the metrics and analyses produced by MAQAO tools. Then, we will explain the choice of the selected compiler structure. Finally we will show how ASSIST can benefit from its integration into MAQAO and vice versa.

3.1 Overview

ASSIST is an open source FDO tool and framework based on the Rose [28] compiler infrastructure and integrated into the MAQAO [3, 23] toolset. More details are provided in the next subsections.

Figure 1 presents an overview of the steps involved in the tool's operation. The following section will provide examples illustrating this process.

ASSIST provides users with a simple yet flexible interface that offers two alternative approaches to specify transformations. The first one makes use of directives while the second one is based on a (Lua) script (depicted as *Transformation script*). he latest provides a means to completely hide the transformations. For example, the directive *!DIR$ MAQAO UNROLL=4* above a loop triggers the unroll (factor of 4) of its body, if applicable and by running the following command: *maqao s2s -option="apply-directives" -src=foo.f90* the transformed code can still be compiled and even reviewed by the programmer if necessary. The source code is parsed and transformed into an abstract syntax tree (AST) that ASSIST will transform accordingly into a given set of directives or a script file. Leveraging optimization opportunities is possible when feedback data from MAQAO [3, 23] is available. For example, to apply the loop count transformation (described in the next section), it is possible to run *maqao s2s -vprof_xp=/path/to/vprof.csv -bin=binary*. It will use MAQAO API

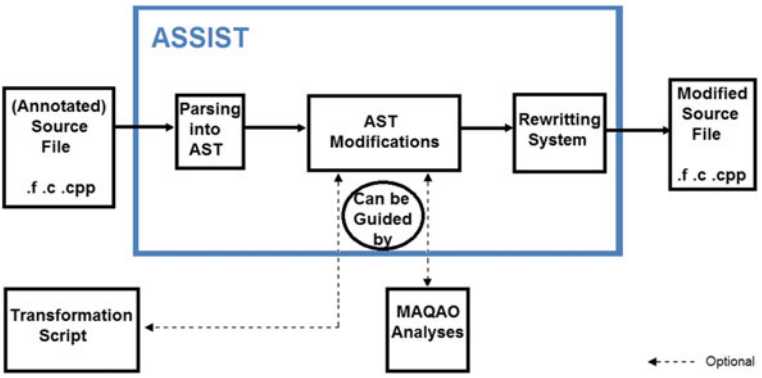


Fig. 1 Overview of the tool's usage

to search for information about loops and files to handle them and read MAQAO VPROF results to apply the loop count transformation. Available analyses are based on MAQAO CQA (code quality) and MAQAO VPROF (value profiling). Finally, the modified AST is parsed to generate a modified source file as output.

3.2 Compiler Infrastructure

Applying transformations to a given source code requires a set of frontends. In our case we will give priority to scientific applications (HPC field), hence selecting C, C++ and Fortran languages. We want an output code that remains at source level and not in a compiler-specific intermediate representation. That is why we chose to code our transformation through the manipulation of an AST.

For all these reasons we decided to look for an existing infrastructure instead of implementing a new one.

There are many compiler available infrastructures and specialized source-to-source frameworks, but only very few can satisfy our requirements. LLVM [11] is a compiler infrastructure that allows the manipulation of an AST through a library. However, it only supports C/C++ languages through Clang. Clang is very useful and easy to use to analyze an AST and add passes to the compiler but not for performing source-to-source transformations. Even if theoretically possible, it is impossible in reality due to a lack of documentation and specialized functions. Transformations are expected at the IR level. Also there is currently no production Fortran support. Very recently Flang [16] was introduced as the new Fortran frontend but it is still in its early phase of development.

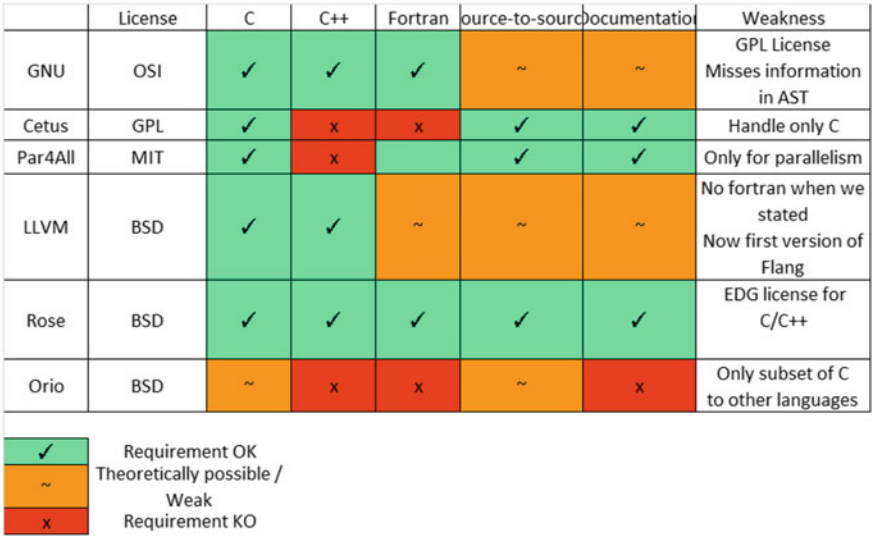
Cetus [15] is a compiler infrastructure featuring source-to-source transformation of C AINSI codes only. DMS [10] is a commercial program analysis and transformation system. That is why it is not included in our comparison table.

Despite some shortfalls (refer to ASSIST's git repository) in the management of the Fortran language that we have managed to overcome, we chose Rose [28]. It is the most suitable framework given our requirements. It is the only open source and easy-to-use (i.e. documented) tool capable of manipulating the AST of C, C++ and Fortran source codes.

Figure 2 presents a summary of the main compiler infrastructures and specialized source-to-source frameworks. The table lists the requirements and how they are fulfilled or not. As we mentioned earlier, Rose appears to suit our constraints best.

3.3 Integration into MAQAO

ASSIST is a MAQAO module. That means that it has access to the MAQAO core (binary and analysis layers) and can also communicate with other MAQAO tools through an internal API. MAQAO tools deal with binary function and loop objects.



	License	C	C++	Fortran	ource-to-sourc	ocumentatio	Weakness
GNU	OSI	✓	✓	✓	~	~	GPL License Misses information in AST
Cetus	GPL	✓	x	x	✓	✓	Handle only C
Par4All	MIT	✓	x		✓	✓	Only for parallelism
LLVM	BSD	✓	✓	~	~	~	No fortran when we stated Now first version of Flang
Rose	BSD	✓	✓	✓	✓	✓	EDG license for C/C++
Orio	BSD	~	x	x	~	x	Only subset of C to other languages

✓	Requirement OK
~	Theoretically possible / Weak
x	Requirement KO

Fig. 2 Constraints array

Since ASSIST manipulates source code it must perform a mapping between real source lines and sources lines provided by the compiler through debug information. That way, ASSIST can establish a link between source and binary functions/loops. This implementation also allows other MAQAO tools to take advantage of ASSIST's ability to analyze and manipulate source code. That is how we extend MAQAO's ability to deal with source code.

The current implementation of ASSIST uses three MAQAO modules: LPROF for profiling (hotspots); CQA for code quality metrics (e.g. vectorization ratio); VPROF for function and loop value profiling. In this paper we only mention the features that are used by ASSIST.

4 Supported Transformations

ASSIST features different kinds of transformations, from common ones like loop unroll to less common ones like loop and function specialization. We did not find any available tools providing such transformations. Moreover these specialization transformations have been specifically designed to be combined with the other available transformations. Block vectorization and loop count transformations are only available in ASSIST.

4.1 Common Loop Transformations

The current implementation of ASSIST supports the following common loop transformations: interchange; unroll (including full unroll); strip mine; tile. Other ones may be added in the future.

4.2 Constant Propagation and Local Dead Code Elimination

Since we can apply multiple transformations we need a means to clean up transformed code eliminating useless chunks generated by specialization (e.g conditionals). For that purpose we implemented constant propagation and local dead code elimination.

After the constant propagation, we browse the AST to check all conditional branches. If a loop is detected with only one iteration, the loop is replaced by its body and the iteration variable replaced by its value in the whole body. ASSIST also checks "if" statements, by checking if the conditional expression is always true or false to replace the whole "if" statement by its "then" body or by its "else" body. To check if a conditional expression is always true or false, the expression is statically evaluated. If it is composed of two integers, we compare them with the corresponding operator. If it implicates a variable, ASSIST tries to trace back through previous assignment statements involving the variable to check if it ends up as a constant and if this assignment is not the result of an "if" condition or a loop. If all of the conditions are true, the variable will be considered as its value and the test continues.

4.3 Specialization

Specialization is the act of creating particular versions of the same code by explicitly considering specific values of one or more variables. For instance we can specialize a loop based on special values of the induction variable. Traditionally we want to handle a loop differently depending on whether it executes a low or a high number of iterations.

Specialization is not an end in itself but just a means to make optimizations happen. It is used when possible to simplify in some way a portion of code based on the knowledge of one or multiple values and their occurrences. As a consequence, the main drawback of specialization is that it can worsen performance if not used sparingly. To perform either loop or function value profiling we rely on MAQAO VPROF. Our specialization transformations can be categorized into two transformations; loop specialization and function specialization.

In the case of function specialization we will usually want to target specific value combinations. Figure 3 provides such an example. A new specialized function is created and the according conditionals are generated. To try to simplify the specialized code we apply our partial dead code elimination pass.

```
#pragma MAQAO SPECIALIZE(N=4,s={1,10})
void foo (int N, int* a, int* b, int s)
{
  int e = s - 10;
  if (e > 20) {
    for (int i=0; i < N; i++) {
      a[i] = b[i];
    }
  } else if (s > 10) {
     for (int i=0; i < N; i++) {
       a[i] -= b[i];
     }
  } else if (s <= 10) {
    for (int i=0; i < N; i++) {
      a[i] += b[i];
    }
}
```

(a) Before function specialization

```
void foo (int N, int* a, int* b, int s)
{
  int e = s - 10;
  if ((N==4)&&(s>0)&&(s<11)) {
   return foo_ASSIST_Ne4_sb0_11(a,b,s);
  }
  if (e > 20) {
    for (int i=0; i < N; i++) {
      a[i] = b[i];
    }
  } else if (s > 10) {
     for (int i=0; i < N; i++) {
       a[i] -= b[i];
     }
  } else if (s <= 10) {
    for (int i=0; i < N; i++) {
      a[i] += b[i];
    }
  }
}

void foo_ASSIST_Ne4_sb0_11_ei11 (int* a,
                        int* b, int s)
{
  int e = s - 10;
  for (int i=0; i < N; i++) {
    a[i] += b[i];
  }
}
```

(b) After function specialization

Fig. 3 Example of function specialization performed by ASSIST

It is possible to apply as many specialization directives as combinations we target. Figure 8 in Sect. 5 is an illustration of such a case. In the current implementation specialization is limited to only integer variables.

4.4 Loop Count Transformation

We saw that loop specialization required an *a priori* knowledge of loops' bound value. This piece of information can be exploited in another way. Intel compilers offers the ability to specify a *loop count (min, max, avg)* directive. The compiler can then make that information available to its optimization passes. By default the compiler will generally generate multiple variants (e.g. scalar, SSE, AVX, etc.) of the same source loop at the binary level. However it will generate much fewer variants by considering loop count data. Helping the compiler in this way throughout the whole application can provide a significant performance gain (see Sect. 5).

```
#pragma MAQAO BLOCKVECB
for (int i=0 ; i < 7; i++ ) {
  a[i] += b[i]
}
```

(a) Before

```
#pragma simd
#pragma vector unaligned
for (i = 0; i < 4; i++) {
  a[i] += b[i]
}

#pragma simd
#pragma vector unaligned
for (i = 4; i < 6; i++) {
  a[i] += b[i]
}
a[6] += b[6]
```

(b) After

Fig. 4 Example of Block Vectorization on x86_64 peformed by ASSIST

4.5 Block Vectorization Transformation

We noticed on some occasions that even when the loop bound was hard-coded the compiler would not vectorize that loop properly. We can check such cases thanks to MAQAO CQA which offers vectorization metrics. This transformation performs the following steps on a given loop: force the compiler to vectorize the loop using *SIMD* directive; prevent peeling code from being generated using *vector unaligned* directive; and adapt the number of iterations to the vector length. Figure 4 illustrates this transformation.

5 Experiments

In this section we will compare our results with the Intel compiler *pgo* mode that we will refer to as IPGO. Intel compilers are neither open source nor free, but they are available on almost all the HPC clusters and provide better performance in our tests (compared to GCC and LLVM). The main reason behind this choice of *pgo* comparison lies in the lack of FDO tools available for regular users. Our goal is not to mimic the *pgo*, rather to present a complementary approach which goes beyond observed limitations.

All the measurements presented below were gathered on an Intel(R) Skylake SP based machine (Intel Xeon Platinum 8170 CPU@2,10GHz) with the Intel compiler

version 17.0.4. Multiple executions (31) were performed to reach statistical stability and avoid outlier measurement data.

Also, this section presents the experimental results of the transformations offered by ASSIST based on feedback data and user insights.

Application Pool

Three functional industrial applications were used to test our approach: Yales2 [14], AVBP [30] and ABINIT [17].

YALES2 is a numerical simulator of turbulent reactive flows using the Large Eddy Simulation method. It is a finite volume code for unstructured meshes, with an innovative 4th order spatial scheme for the discretization of convective and diffusive terms. It is based on the low-Mach number approximations of the Navier-Stokes equations, which solves an elliptic Poisson equation at each iteration and scales well to over 16K cores. The MPI version uses subdomain decomposition with adjustable domain size, allowing efficient cache usage. ASSIST has been tested on two of their datasets named "3D_cylinder" and "1D_COFFE". The application is written in Fortran 2003.

AVBP is parallel CFD code developped by CERFACS that solved the three-dimensional compressible Navier Stokes equations on unstructured multi-element grids. It uses third space and time Taylor Galerkin numerical schemes. The code has been ported and tested up to 200K cores with an 85% strong scaling efficiency (BG/Q) for a 200M element case (1000 elements per MPI rank). Cache coloring uses the reverse Cuthill-Mckee method. ASSIST has been tested on two representative datasets names SIMPLE (helicopter chamber demonstrator combustion simulation) and NASA (NACA blade simulation). The application is written in fortran 95.

ABINIT is a package allowing users to find the total energy charge density and electronic structure of systems made of electrons and nuclei (molecules and periodic solids) within Density Functional Theory (DFT) using pseudopotentials (or PAW atomic data) and a planewave basis. The application is developped in Fortran 90.

Impact of Loop Value Profiling

Our first FDO optimization is based on knowing loop trip counts obtained by value profiling using MAQAO VPROF. When loops exhibit a complex control flow due to multi-versioning, knowing the trip count can help the compiler simplify the decision tree. We will refer to the loop count transformation as LCT for the remaining part of this section.

Figure 5 presents the speedups obtained with LCT, IPGO and the combination of both for each application/dataset. Both LCT (15%) and IPGO (14%) provide a performance gain for the Yales2 using 3D_cylinder as a dataset. The combination of both LCT and IPGO raises the gain to 19%. For the second Yales2 dataset (1D_COFFE) both endeavors only reach 5%. However, the combination does not pay off. For AVBP, running the SIMPLE data set, we observe a negligible speedup. However, for AVBP individual contributions of IPGO and ASSIST can be partially

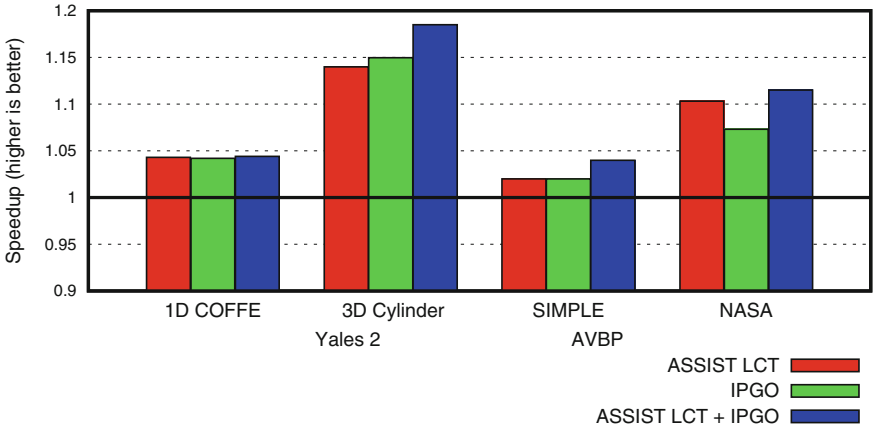


Fig. 5 Impact (speedup) of ASSIST LCT when compared to IPGO

combined. IPGO provides a 10% speedup on AVBP with the NASA dataset while LCT only achieves 7%. The combination reaches 12%.

This study shows that providing the compiler with loop trip count feedback (minimum, average and maximum values) results in a performance gain. We can also observe that the combination with *pgo* can lead to a higher gain.

Specialization

While optimizing applications, we noticed that we often resort to function and/or loop specialization before applying other transformations. The following two examples show how coupling specialization with other transformations can provide significant performance gain.

AVBP

In this example we couple both specialization and block vectorization transformations applied to the ten most time-consuming functions. We first apply loop and function specialization separately, then we apply block vectorization on the most efficient version. We also apply the LCT and the IPGO on the original version to verify whether the compiler is able to perform better using additional guidance.

Figures 6 and 7 compare the speedup ratios of each version with the original one. Function and loop specialization are performed separately and presented here to show their individual impact.

We observe that block vectorization can offer a 2.6x performance gain while the loop and function specialization only achieve, at best, a speedup of 1.5x. Performing only loop or function specialization may be counterproductive in some cases because of the induced complexity of the control flow if no further induced optimizations are possible.

When the compiler fails to vectorize a loop properly, the block vectorization transformation is very effective given that it explicitly exposes a simpler loop structure

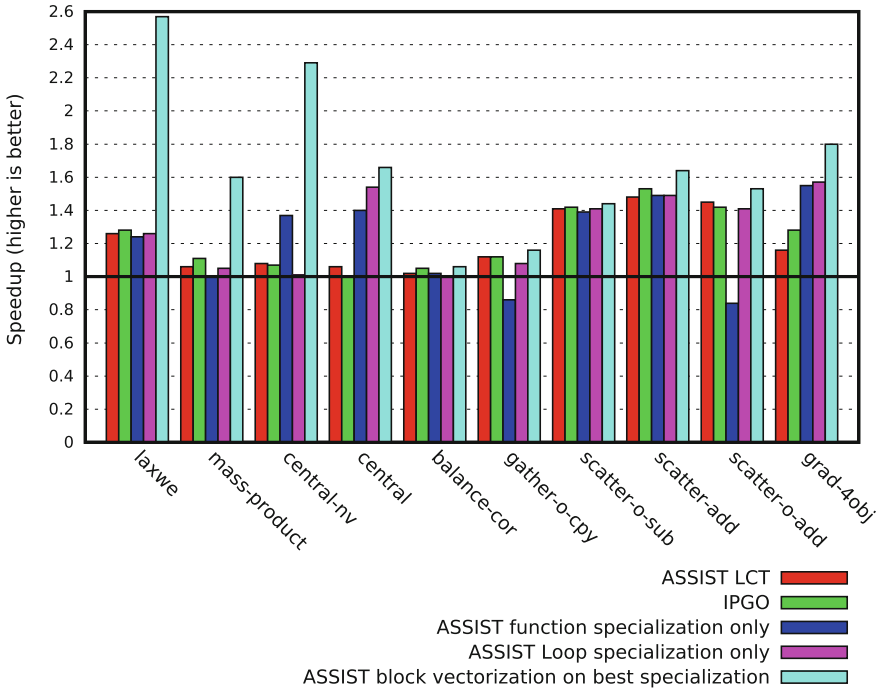


Fig. 6 AVBP using the SIMPLE dataset—Speedups by function before and after applying transformations with ASSIST (block vectorization, function/loop specialization, LCT) and IPGO compared with the original version (Higher is better)

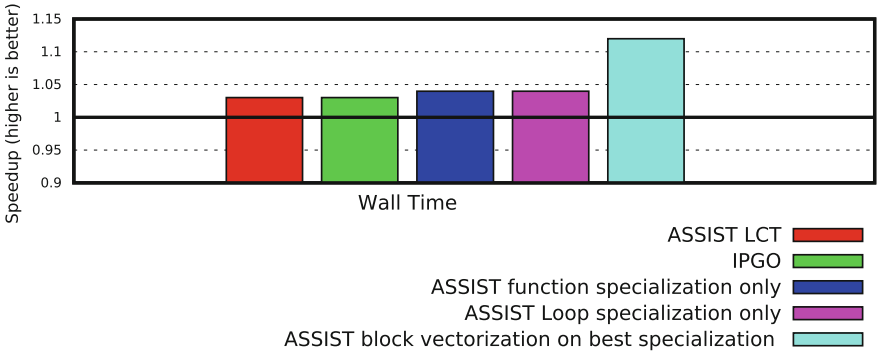


Fig. 7 AVBP using the SIMPLE dataset—Speedups of the wall time before and after applying transformations with ASSIST (block vectorization, function/loop specialization, LCT) and IPGO compared with the original version (Higher is better)

with no peel or tail loops to the compiler. In our case, we can evaluate the vectorization ratio of a loop using MAQAO CQA; ASSIST can automatically trigger the transformation from the CQA results by extracting several items of information, like the vectorization ratio metric, the file and the function where the loop is. The block vectorization transformations force the compiler to vectorize small loops with a small number of iterations; the compiler also fully unrolls these loops.

ABINIT

In this example, ASSIST is fully driven by the user. At first, a full profiling of the code is performed, followed by value profiling on one of the main hotspots of the application. Three input parameters were found to be of importance.

First, the function can be called with two different types of input data, either real-valued data or complex-valued data. A given test case will almost exclusively use one or the other. As those data are expressed as an array with one or two elements in part of the code, specialization of this value simplifies address computations and vector accesses by making the stride a compile-time constant rather than a dynamic value.

Second, multiple variants of the algorithm are implemented in the function. Which exact variant is used depends on two integer parameters. Again, a given test case is usually heavily biased toward a small subset of possible cases. Specialization to one case removes multiple conditionals. As the loop nests for a given case appear in different branches, this removal of conditionals exposes the true dynamic chaining of loop nests to the compiler with no intervening control flow break.

Once specialized with ASSIST, the function becomes much simpler to study. It turns out that the dominant loop nest in the function is amenable to loop tiling. A large array is updated in its entirety inside a loop, a bad pattern for cache usage. Loop tiling make it possible to updates the array by block, and to only scan and update the array once. While this work would not be particularly difficult to do by hand, more than two dozen variants of the loop nest with similar properties appear in the original function. As the transformed loop adds an extra loop to the nest, complicates indices, and requires a remainder loop, it is much easier and much more reliable to automate the transformation process.

Figure 8 shows the directives on an extract of the function, in part (a). Three specialized variants are produced for the common use cases in our reference test Ti256, by the first three lines of the figure. The critical loop nest is subsequently tiled, but only in the specialized version, by the directive immediately above the loop nest. Part (b) show extracts from the output of ASSIST. The original function now calls the specialized variants whenever the parameters are appropriate. Every conditional previously dynamically encountered is now collapsed into that one test. Below the original function, Fig. 8 also shows the new loop nest with the loop tiling transformation applied. Only 8 elements (a friendly value for a vectorizer) are computed in the innermost loop versus the entire array previously. An outer loop has been added which scans the entire array by block of size 8. In practice, the innermost loop is removed by the compiler, which fully unrolls and vectorizes it.

```
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt=3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt<3,cplex=2)
!DIR$ MAQAO SPECIALIZE(choice=1,paw_opt>3,cplex=2)
subroutine opernlb_ylm(choice,cplex,paw_opt,...)
  ...
  if (choice == 1) then
    !DIR$ MAQAO TILE_INNER_IF_SPE_choicee1=8
    do ilmn=1, nlmn
      do k=1,npw
       ztab(k) = ztab(k)+ffnl(k,1,ilmn)*cmplx(gxfacs_(1,ilmn),gxfacs_(2,ilmn),kind=dp)
      end do
    end do
  end if
  ...
end subroutine
```

(a) Before ASSIST transformations

```
SUBROUTINE opernlb_ylm(...)
  IF ((choice.EQ.1).AND.(paw_opt.EQ.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
    RETURN
  END IF
  IF ((choice.EQ.1).AND.(paw_opt.LT.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opti3_cplexe2(...)
    RETURN
  END IF
  IF ((choice.EQ.1).AND.(paw_opt.GT.3).AND.(cplex.EQ.2)) then
    CALL opernlb_ylm_ASSIST_choicee1_paw_opts3_cplexe2(...)
    RETURN
  END IF
 ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opte3_cplexe2(...)
  ...
  lt_bound_npw = (npw / 8) * 8
  DO lt_var_k = 1, lt_bound_npw, 8
    DO ilmn = 1, nlmn
      DO k = lt_var_k, lt_var_k + (8 - 1)
       ztab(k) = ztab(k)+ ffnl(k,1,ilmn)* cmplx(gxfacs_(1,ilmn),gxfacs_(2,ilmn),kind=dp)
      ENDDO
    ENDDO
  ENDDO
  ...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opti3_cplexe2(...)
...
END SUBROUTINE
SUBROUTINE opernlb_ylm_ASSIST_choicee1_paw_opts3_cplexe2(...)
...
END SUBROUTINE
```

(b) After ASSIST transformations

Fig. 8 ABINIT—Example of function specialization coupled with loop tiling, performed with ASSIST, for the use case Ti-256. Boxes highlight the tiling transformation of the innermost loop

Speed up results are shown in Fig. 9. The original version is at one by definition. We added IPGO to show the potential of our approach. Specialization offers a small gain but the dominant issue is still the time spent in the critical loop nest. Adding tiling offers a large gain of almost 1.8x in total by significantly reducing the memory

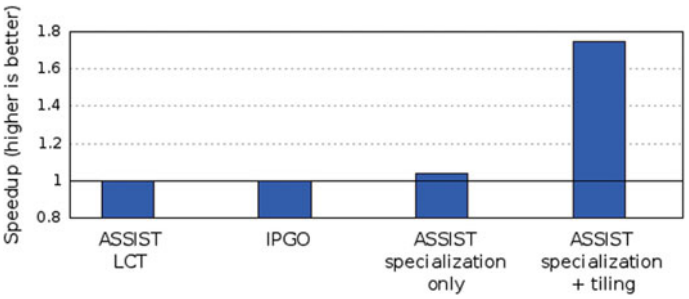


Fig. 9 ABINIT—Ti-256—Speedups of IPGO, ASSIST LCT, specialized with ASSIST, specialized and tiled with ASSIST compared to the original version

bandwidth usage of the critical loop nest. Despite the complexity of the original function, ASSIST would make it easy to apply the same transformations to other possible use cases of the function for other test cases of the ABINIT code.

6 Related Work

The originality of the approach presented in this paper lies in the combination of both source-to-source transformations using annotations and FDO approaches. More precisely feedback data drives source-to-source transformations to achieve both productivity and performance.

Orio [19] is the closest tool and approach to ASSIST. We share the same goals, namely improving productivity and performance using annotations at source level as well as being able to handle architecture-specific/independent code optimizations. However, they use empirical performance tuning to achieve better performance. This implies generating multiple variants and evaluating their cost. Our approach opts for a cheaper and more straightforward path using FDO. Our approach encompasses static and dynamic analyses. This means that we can assess the quality of the code generated by the compiler (using MAQAO CQA [29]) and get execution behavior metrics. CHiLL [12] is a framework that provides loop level transformations and also uses empirical optimizations. It targets compilers and not regular developers. Xevtgen [31] goes a step further when considering source-to-source transformations. It allows application developers to define their own transformations using a dummy Fortran syntax coupled with directives. From our own experience in helping developers optimize their code, we can claim it is dangerous to assume they will be willing to invest time and ressourses to write their own transformations, even if the interface is based on a well-known language such as Fortran. For this particular reason, we have tried to provide as many predefined transformations as possible. Also, plenty of Domain Specific Languages or frameworks are available for performing source-

to-source transformations, i.e: [10, 13, 27, 33, 36]. Some also implement parallel transformations [1, 6, 21, 25, 26, 32].

For FDO, the related work analysis is straightforward: there are very few tools and the main goal is to achieve performance. From what we encountered during our research, the only available tools implementing FDO are compilers, with PGO (e.g. Intel, GCC, LLVM), and AutoFDO (e.g. Intel [20], GCC [18] and LLVM) modes. AutoFDO [9] is also the name of an in-house FDO deployment system proprietary to Google. Compared to PGO, AutoFDO exploits hardware counter profiles. In both cases feedback/profile data are injected early in the intermediate representation of the compiler so that all the optimization passes can take advantage of them. Our approach aims to help modern compilers by not injecting data using a specific format, but rather at source level. From a performance point of view, both approaches are complementary. During our work, we also came across Aestimo [2], an FDO research evaluation tool that can be coupled with the Open Research Compiler [34]. However it does not pursue the same goals.

7 Conclusion and Future Work

ASSIST is an open source tool that was developed with the aim of providing assistance to application programmers in order to achieve better productivity and code performance. We have shown the effectiveness of our approach when dealing with industrial applications by using either static and dynamic feedback data, or user guidance.

The tool presented in this article provides the foundation for an autotuning tool. As future work we plan to harness all the available dynamic analyses existing in MAQAO including those using hardware counters to perform and automate more optimizations.

Acknowledgements We would like to thank Gabriel Staffelbach (CERFACS) for providing our laboratory with the AVBP application, as well as Ghislain Lartigue and Vincent Moureau (CORIA) for providing us with YALES2. This work has been carried out by the Li-PaRAD laboratory, PeXL and the Exascale Computing Research laboratory, with the support of CEA, Intel, UVSQ. Intel granted us dedicted access to a Skylake SP machine on which the experiments were run. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the CEA, Intel, or UVSQ.

References

1. Advisor. https://software.intel.com/en-us/intel-advisor-xe
2. Amaral, J.N., Berube, P.: Aestimo: a Feedback-Directed Optimization Evaluation Tool. IEEE, Piscataway, NJ, USA (2006)

3. Barthou, D., Rubial, A.C., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 openmp codes with maqao. In: Parallel Tools Workshop, pp. 95–113. Desden, Germany, September 2009. Springer
4. Bendifallah, Z., Jalby, W., Noudohouenou, J., Oseret, E., Palomares, V., Rubial, A.C.: PAMDA: performance assessment using MAQAO toolset and differential analysis, pp. 107–127. Springer International Publishing, Cham (2014)
5. Bodin, F., Dolbeau, R., Bihan, S.: Hmpp: a hybrid multi-core parallel programming environment. In: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007), vol. 28 (2007)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Notices, pp. 101–113. ACM (2008)
7. Charif Rubial, A.S., Lereste, J.-B.: https://www.maqao.org/release/MAQAO.Tutorial.LProf.v1.pdf
8. Charif-Rubial, A.S., Barthou, D., Valensi, C., Shende, S., Malony, A., Jalby, W.: Mil: a language to build program analysis tools through static binary instrumentation. In: 20th Annual International Conference on High Performance Computing, pp. 206–215, Dec 2013
9. Chen, D., Xinliang Li, D., Moseley, T.: Autofdo: automatic feedback-directed optimization for warehouse-scale applications. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, pp. 12–23. ACM, New York, NY, USA (2016)
10. Chris Lattner et Vikram Adve. Dms/spl reg: program transformations for practical scalable software evolution. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 625–634. IEEE (2004)
11. Chris Lattner et Vikram Adve. Llvm a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p. 75. IEEE Computer (2004)
12. Chun Chen, J.C., Hall, M.: Chill: a framework for composing high-level loop transformations, June 2008
13. Cordy, J.R.: Source transformation, analysis and generation in txl. In: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2006, pp. 1–11. ACM, New York, NY, USA (2006)
14. Coria. http://www.coria-cfd.fr/index.php/YALES2
15. Dave et al.: Cetus: a source-to-source compiler infrastructure for multicores. Computer, 36—42, December 2009
16. flang. https://github.com/llvm-flang/flang
17. Gonze, X. et al.: Abinit: first-principles approach to material and nanosystem properties. Comput. Phys. Commun., 2582–2615. Elsevier (2009)
18. Google. https://github.com/google/autofdo
19. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using orio. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp. 1–11, May 2009
20. Intel. https://github.com/google/autofdo
21. Irigoin et al: Interprocedural analyses forprogramming environments. In: Workshop on Evironments and Tools For Parallel Scientifc Computing, Saint-Hilaire du Touvier, France, August 1992
22. Koliaï, S., Bendifallah, Z., Tribalat, M., Valensi, C., Acquaviva, J.-T., Jalby, W.: Quantifying performance bottleneck cost through differential analysis. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 263–272. ACM, New York, NY, USA, (2013)
23. MAQAO toolsuite. http://www.maqao.org
24. Novillo, D.: Samplepgo: the power of profile guided optimizations without the usability burden. In: Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM-HPC 2014, pp. 22–28. IEEE Press, Piscataway, NJ, USA (2014)
25. Palkowski, M., Bielecki, W.: TRACO Parallelizing Compiler, pp. 409–421. Springer International Publishing, Cham (2015)

26. Paraformance. http://paraformance.weebly.com/
27. Paul Klint, J.V., van der Storm, T.: Rascal a domain specific language for source code analysis ad manipulation. In: IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 168–177. IEEE Computer Society (2009)
28. Quinlan et al.: Rose: compiler support for object-oriented framework. In: Parallel Processing Letters, pp. 215—226. Lawrence Livermore National Laboratory, Livermore, CA, USA, October 2000. World Scientific
29. Rubial, A.C., Oseret, E., Noudohouenou, J., Jalby, W., Lartigue, G.: CQA: a code quality analyzer tool at binary level. In: HiPC, pp. 1–10. IEEE Computer Society (2014)
30. Rudgyard, M., Schonfeld, T.: Steady and unsteady flow simulationsusing the hybrid flow solver avbp. AIAA J., 1378–1385. AIAA ARC (1999)
31. Takizawa, H., Suda, R., Hirasawa, S.: Xevtgen: fortran code transformer generator for high performance scienti c codes. Int. J. Network. Comput., 263—289 (2016)
32. Verdoolaege, S., et al.: Polyhedral parallel code generation for cuda. ACM Trans. Architec. Code Optim. ACM, January 2013
33. Vermaas, R., Bravenboer, M., Kalleberg, K.T., Visser, E.: Stratego/xt 0.17. a language and toolset for program transformation. In: Science of Computer Programming. Elsevier, May 2008
34. Wu, C., Lian, R., Zhang, J., Ju, R., Chan, S., Liu, L., Feng, X., Zhang, Z.: An Overview of the Open Research Compiler, pp. 17–31. Springer, Berlin Heidelberg, Berlin, Heidelberg (2005)
35. Xiao, X., Hirasawa, S., Takizawa, H., Kobayashi, H.: An approach to customization of compiler directives for application-specific code transformations. In: 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, pp. 99–106, Sept 2014
36. Yi, Q.: Poet: a scripting language for applying parameterized source-to-source program transformations. In: Software Practice And Experience, pp. 675–706. University of Texas at San Antonio, USA, May 2012. John Wiley and Sons

Unifying the Analysis of Performance Event Streams at the Consumer Interface Level

Jean-Baptiste Besnard, Allen D. Malony, Sameer Shende, Marc Pérache, Patrick Carribault and Julien Jaeger

Abstract Several instrumentation interfaces have been developed for parallel programs to make observable actions that take place during execution and to make accessible information about the program's behavior and performance. Following in the footsteps of the successful profiling interface for MPI (PMPI), new rich interfaces to expose internal operation of MPI (MPI-T) and OpenMP (OMPT) runtimes are now in the standards. Taking advantage of these interfaces requires tools to selectively collect events from multiples interfaces by various techniques: function interposition (PMPI), value read (MPI-T), and callbacks (OMPT). In this paper, we present the unified instrumentation pipeline proposed by the MALP infrastructure that can be used to forward a variety of fine-grained events from multiple interfaces online to multi-threaded analysis processes implemented orthogonally with plugins. In essence, our contribution complements "front-end" instrumentation mechanisms by a generic "back-end" *event consumption interface* that allows "consumer" callbacks to generate performance measurements in various formats for analysis and transport. With such support, online and post-mortem cases become similar from

J.-B. Besnard (✉)
ParaTools SAS, Arpajon, France
e-mail: jbbesnard@paratools.fr

A. D. Malony · S. Shende
ParaTools Inc., Eugene, USA
e-mail: malony@paratools.com

S. Shende
e-mail: sameer@paratools.com

M. Pérache · P. Carribault · J. Jaeger
CEA, Arpajon, France
e-mail: marc.perache@cea.fr

P. Carribault
e-mail: patrick.carribaul@cea.fr

J. Jaeger
e-mail: julien.jaeger@cea.fr

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_4

an analysis point of view, making it possible to build more unified and consistent analysis frameworks. The paper describes the approach and demonstrates its benefits with several use cases.

1 Introduction

There has always been an intimate *pas de trois* between the need for greater computational power in scientific discovery, the evolution of HPC hardware and software to provide next-generation computing potential, and the nature of parallel performance on HPC platforms that determines what can be achieved. By now, the dance is well-known. New domain applications try to maximize parallelism and handle larger problems, pushing beyond the limits of present high-performance computing (HPC) capabilities. In response, HPC architectures and technology advance, resulting in greater programming complexity in order to access the potential these new HPC machines have to offer. However, the performance complexity is also in flux. For instance, a parallel application might be able to expose a high level of concurrency for a large number of threads to execute on a many-core processor. The problem is that the optimal number of threads to use and even cores to allocate at any point in the program depends on the performance interactions involved. The interplay of the application and the HPC resources with respect to performance factors is subtle and not easily understood. Performance consequences include:

- a decreasing memory per thread jeopardizes a pure distributed memory model due to both the memory replication (halo cells) and communication overhead [4];
- smaller cores result in lower sequential performance, requiring the programs to exhibit more parallelism to achieve the same performance;
- larger vectorial units requiring optimizing compilers and a correct expression of computing loops (making them *vectorizable*), possibly new optimized instructions; and
- complex memory hierarchies demanding a careful data-placement and tracking (first-touch), including important NUMA effects, stacked memory, deep cache architectures and meshed processors.

A strategy in response to these complexity challenges has been to advance parallel programming languages to expose sufficient parallelism that can be mapped to multiple shared-memory cores via multi-threading and to distributed nodes via message passing. Unified programming abstractions and hybrid ones are commonly explored. The hybrid approach has been the most widely adopted due to the progressive shift it allows from a pure MPI program to an *MPI + X* one. In general, X is some form of shared-memory parallelism, such as provided by OpenMP directives for parallelizing loops or expressing tasks. The standardization and portability of MPI and OpenMP, plus the large corpus of multi-million line programs, makes it a desirable combination. In general, hybrid parallel programming methodologies attempt to deal

with the important problem of application engineering in the face of rapid hardware evolutions. The hope is to minimize the impact on the code which is costly to develop and often represents a multi-year investment spanning over machine generations.

However attractive this strategy is, it is incomplete because performance aspects are not fully considered. Beyond providing a means to enable instrumentation, most parallel programming systems do not provide direct support for performance analysis or optimization. Certainly, parallel performance tools exist, but there is a strong motivation to make sure performance technology can interoperate effectively. Interestingly, the rise of hardware targeted to HPC will make this more of a challenge. Dedicated ARM processors in Japan and the Sunway processors [10] in China increase the number of architectures tools will need to support. Similarly, custom high-performance networks such as the Tofu interconnect [2] in Japan and the Bull Exascale Interconnect [7] in France add communication performance factors specific to those platforms. Infrastructure and power monitoring with the Bull power-tracking FPGA results in additional components needing to be integrated. All these evolutions will need (often already lack) suitable tools to both allow and assess them. Yet, the shift to such systems will likely require a substantial porting effort, which will be further complicated by the dearth of performance tools.

The purpose of this paper is to look at the problem of tool interoperability and propose an interface that could be leveraged more efficiently to address certain complexity challenges of HPC hardware and software evolution. In particular, we consider tools with complementary instrumentation mechanisms and pose the open question of how to couple their data streams for measurement and downstream processing. Indeed, several leading performance tool systems have *instrumentation chains* that are redundant, while their actual value comes from the analysis insight they provide. Thus, we propose a component-based view of the *instrumentation chain* by defining a coupling at the consumer-level interface instead of at the transport-level, as is commonly done.

To support this idea, the rest of the paper is organized as follows. Section 2 first describes our component-based instrumentation chain and how it is articulated. The feasibility, extensibility, and a possible implementation of our proposed common consumer interface is discussed in Sect. 3. Section 4 illustrates an instance previous ideas in the context of the Multi-Application Online Profiling tool (MALP). Concluding remarks and future directions are then given.

2 Components in the Instrumentation Chain

The typical parallel program *instrumentation chain* focuses on "observation" points and the mechanisms to make these visible, versus potential "coupling" opportunities for interoperability. Our goal is to determine where coupling interfaces could be enabled exist and how to support them. To do so, we will first have to describe a general model of the instrumentation chain and then consider each of its components. With this groundwork, we propose a consumer-level interface and discuss its prototype implementation.

2.1 General Model

The *instrumentation chain* is a general term describing all the steps needed by a tool to obtain and forward a target program state to intelligible analysis consumers, thereby providing the end-user with a clear view of what is the actual behavior of its program. This model applies for debugging, validation, and performance assessment with only little variations.

Figure 1 shows the major components in the instrumentation chain that are used to build tools. From left to right, we first have the *event-source* where the event is generated, for example, by a function called or being called. Second, the data associated with what was observed at the source is structured as an *event* represented by different fields encoding observable parameters. In the middle of the figure, these events are *transported* to the analysis. This process can take several forms and possibly converts the event to an intermediate representation suitable for the transport layer. Then, on the analysis side, the transported event is first decoded to a state similar to the original *event* representation. Eventually, events are projected to the final *analysis* before being presented to the end-user.

The instrumentation chain shown in Fig. 1 is the way support tools explore parallel program state, systematically intercepting, encoding, forwarding, decoding, and eventually projecting what happened in the program. Events by nature are imperceptible, thus requiring this chain to be actualized for analysis. Also, note that we do not see this chain as exclusively feed-forward. It is also possible for the end-user to *consume* events from the event source in a "pull" model. In this case, for example, when considering a debugger, the analysis is directly consuming events through the *ptrace* transport layer.

Let us further describe the components that we have just introduced in more detail. This gives us the opportunity to also point to the rich related work implementing these various building blocks.

2.2 Event Sources

The *event source* is where the observable is generated. In all cases, the goal is to extract information from program's state, but this may happen in many different forms. There are several possibilities when it comes to instrumenting a parallel program [23], which we attempt to sort in categories before discussing them more generally. Expert tool developers are quite familiar with these methods.

Fig. 1 Macroscopic view of instrumentation chain building blocks

- **Manual instrumentation** is the process of manually adding probes in the target program. It can be, for example, tools call outlining phases, perhaps using *printf* to output arbitrary values. Moreover, the user may expose an interface to make visible the state of its program.
- **Source-to-source instrumentation** consists of parsing the source code in order to insert probes and writing out the instrumented code in a source form to be compiled. This was the case for the Opari instrumenter [20] before the advent of the OMPT [8] interface. Included in this category are preprocessors that define redirecting calls to instrumented libraries.
- **Compiler instrumentation** can be used to instrument functions using directives such as `-finstrument-functions`. Moreover, pragmas or functions attributes can be used to create weak symbols to be used at link or load time, as done for example in the MPI profiling interface. An instrumented program is produced as the result of the compilation process.
- **Linker and loader** mechanism can be used to alter symbol resolution either by changing library order at link time, wrapping symbols (using `-Wl,-wrap`) or by inserting at runtime (with `LD_PRELOAD`) a library superseding existing ones. At this level, weak symbols can also be replaced by instrumented functions.
- **Runtime tools** can provide verbose information source for tools. The MPI tools interface with MPI-T [16], the OpenMP tools and debugging interface (OMPT and OMPD) and the CUDA profiling interface (CUPTI) [19] are examples of facilities provided by the runtime to enable tool support.
- **Indirect measurements** in this last category we consider measurements which deal with system-wide parameters such as memory, network bandwidth, system load. This approach can also be extended to interpreted languages. Similarly, when considering virtualized environments or embedded hardware such interface could be exposed through a serial port (either virtual and physical).

The above non-exhaustive list illustrates the number of inputs an instrumentation chain may have to gather. Given the different interfaces offered by the various sources, transposing the "state" of an application to a unified form becomes a challenge. To do so it has to *encode* what it "sees" through these varying sources in a common intermediate representation.

2.3 Intermediate Representations

A tool interested in the various event sources has to ultimately forward data to the analysis consumers. Intuitively, it must convert from the source to what we describe below as an *intermediate representation*. Note that such representation is far from being fixed. Some approaches could adopt actual events, carefully describing what is observed. Other approaches may simply call functions encoding program state as parameters operating the conversion from the source to the transport layer (see next section) immediately in the instrumentation code. Eventually, such representation

could be even useless in the case of a direct data consumption—directly feeding the analysis at wrapper level.

We propose to develop the intermediate representation idea that can serve to create a common intermediate layer between tools components. As we just evoked, we have seen that events can be either partially or even completely diluted in function of the instrumentation chain construction. However, by preserving the "event" abstraction, it is possible to enable interesting scenarios that are not yet fully exploited. Eventually, the reader should notice that this representation is present in two phases of the instrumentation chain (see Fig. 1), both before and after the transport layer.

2.4 Event Forwarding

Given events extracted and then encoded in an intermediate representation, we consider how his representation is transposed to where it should be processed. At this step, there are also various approaches which depend on data verbosity and analysis cost. It is also here where the most impact on tool's scalability arises, clearly of high importance for tool designers. Consider the following transport layers:

- **In-place instrumentation** is when the analysis is directly done at the event-source level. In this case, the analysis is generally not distributed and must remain lightweight so as not to impact the target program.
- **Post-mortem instrumentation** consists in storing events in a trace (generally in the file-system) for later analysis. The advantage of this approach is that analysis is completely decoupled from the target program and that data can be processed multiple times, for example, to be compared. However, the cost associated with storing events has to be mitigated requiring a careful design of a storage medium commonly called a trace-format. Note that this format is generally different from the intermediate representation mentioned in Sect. 2.3.
- **Online instrumentation** is a compromise between the two previous approaches. It relies on network resources to pass events to third-party processes in charge of performing the analysis. This can be done over a Tree-Based Overlay Network (TBON) to perform, for example, validation or continuous spatial reduction at runtime. Another model consists of forwarding events from the group of instrumented processed directly to the analysis—a model that we explore later in this paper in the context of the MALP [5] performance tool.

The purpose of the transport layer is to forward events to the analyzer. In our instrumentation chain model, it can be seen as a function taking events (at the intermediate representation level) and encoding them in a suitable format. On the analysis side, the event is decoded to the intermediate representation for processing. Note, the trace formats also have to account for meta-data (e.g., active threads, their location, their identifiers, and so on). Once forwarded, data are to be projected to the final analysis.

2.5 Event Analysis

There are as many types of event projection than performance tools. Indeed, it is generally at this step that tools produce views, analysis, and hopefully insights for the end-user. Naturally, some views require more input-data than others. However, as analysis are derived from the same input-set of source-event, it is possible to consider that any analysis presented with a sufficiently verbose input should be able to produce its output. In other words, the difference between a time-line and a profile is the level of data projection operated (and therefore its partial destruction). In practice, a given analysis is generally tied to a single instrumentation layer. This might be one reason that we see less attention to interoperability. One interest is to explore whether it is possible to decouple these two main components of the instrumentation chain, enabling cross-analysis.

2.6 Tools Interoperability

We have seen that despite variability in both methods and implementations, instrumentation chains exhibit a common architecture. Moreover, as instrumentation sources are a finite set and generally similar between performance tools, we believe it is reasonable to pose the question of tools interoperability. Some tools are already connected and able to share the same instrumentation layer, such as provided by Score-P [18]. However, this interoperability is still at the transport layer level—Score-P supports several output formats for profiling (TAU [23], Scalasca [12]) and OTF2 [9] for tracing, as well as exposing an online consumption interface for Periscope [3].

Moreover, there are several trace-format converters that can pass data between tools. This process generally involves rewriting all events to the target's transport layer representation (i.e., trace format). This approach has the drawback of duplicating performance data, while being relatively expensive due to their potentially large trace sizes. The following sections explore an alternative approach by exploiting the intermediate representation itself.

3 Towards a Shared-Representation of Performance Events

As presented in Fig. 1, there are two steps where events are in an intermediate representation. During these steps, events are being either extracted or inserted from and to another interface. This representation does not suffer from the same constraints than the transport one. Indeed, these state events are mostly on the stack as structures or function parameters. Also, this representation does not have to be highly space efficient as it will be used for a very short period of time unlike, for example, a post-mortem trace event. An interesting question is whether we can use this to the benefit of unifying represetations.

3.1 Towards a Tool Network

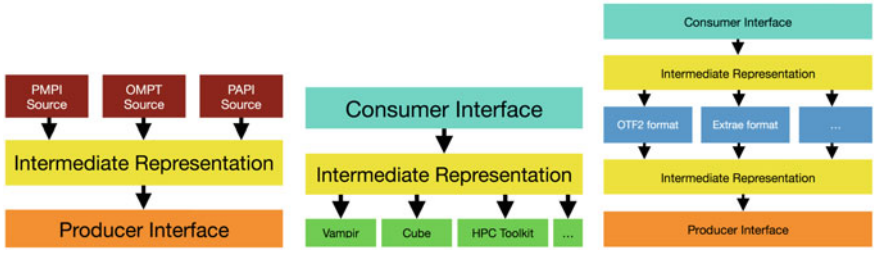
Consider instrumentation chains where this representation is unified. To do so, we suppose a shared intermediate *in-memory* representation sufficiently generic to describe any event type with retro-compatibility. This in-memory aspect is important as it mitigates approach drawbacks. Indeed, if, for instance, we consider the highly generic Pajé [17] trace format, it required ASCII encoding, making it inefficient in terms of storage and parsing.

What we attempt to describe here is close to the consumer/producer interface of any trace format. It can be illustrated with the EARL high-level trace consuming interface proposed in the KOJAK instrumentation chain [25], more recently the notion of Event-Action mapping [15] or ScrubJay [13] (internally relying on Caliper [6]). One point of interest in this related work is the exploration of an event meta-model. Interestingly, in order to make this unified approach possible tools would have to agree on what source events actually represent. If such in-memory event model could be achieved, we could provide trace formats with the same interface. Symmetrically, we could allow instrumentation blocks to produce events in this format.

We propose to model each building block of the instrumentation chain as graph components with inputs and outputs respectively matching these consumer and producer interfaces. As illustrated in Fig. 2, this model would turn support tools into a network of collaborating applications, allowing them to be interfaced at the *consumer interface* level. The following discusses what could be the design of such interface and what are the main constraints envisioned. The approach that we will correlate with our existing implementation of the MALP performance tools which exhibits some of these abstractions.

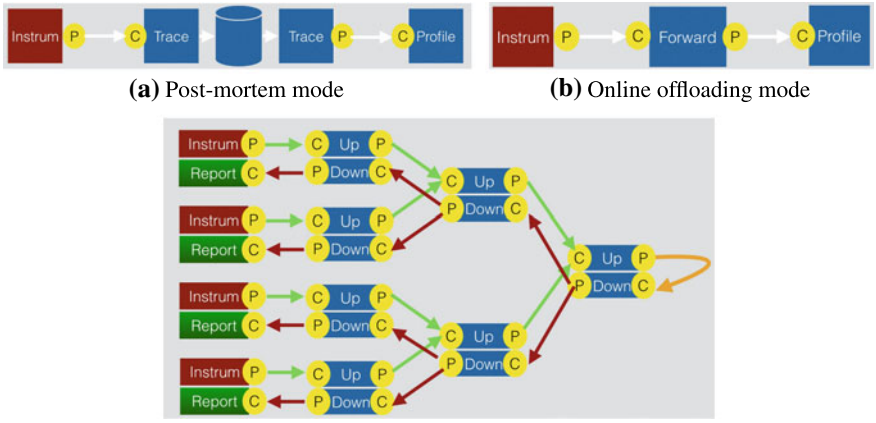
As presented in Fig. 3, the component model supports all the configurations described in Sect. 2.4. Thus, at first cut, this model fits most performance tools indifferently from their data-management policy.

Moreover, as shown in Fig. 4, enabling these common interfaces would allow straightforward interoperability. It would indeed be easy to convert to and from a



(a) Instrumentation block **(b)** Analysis block **(c)** Transport block

Fig. 2 Schematic illustration of *instrumentation chain* building blocks considering the unified producer/consumer model



(a) Post-mortem mode **(b)** Online offloading mode

(c) Tree-Based Overlay Network (TBON) mode

Fig. 3 *Instrumentation chain* configurations obtained from combining building blocks at the consumer level interface. "P" and "C" respectively stand for producer and consumer interfaces

(a) Example of a trace generator **(b)** Implementation of a trace converter

Fig. 4 Alternative building-block usage for interoperability. "P" and "C" respectively stand for producer and consumer interfaces

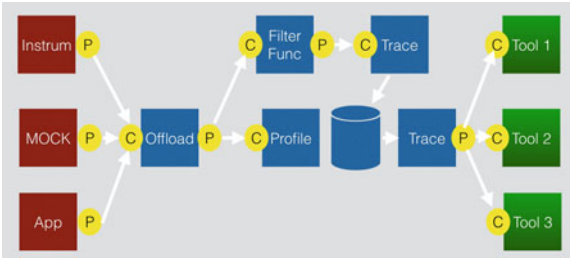


Fig. 5 Custom performance analysis work-flow build around the notion of unified consumer/producer interface. "P" and "C" respectively stand for producer and consumer interfaces

trace-format (Fig. 4), to generate a trace (Fig. 4), or simply to read a trace agnostically from its internal format.

Figure 5 illustrates the level of modularity which could be attained thanks to common interfaces. In this examples, all events are extracted from the source programs using an online coupling (forwarding) mechanism. Then the tool branches, first feeding a selective branch (see *filter func*) sent to the post-mortem trace. Meanwhile, all events are sent to a profile analysis which is less costly to implement and therefore able to handle all events without filtering. Moreover, note how the same trace could be read by different tools. Thanks to this feature users would be able

to see the exact same events instead of having to rerun the instrumented programs will all the bias it supposes (instrumentation overhead). In such configuration, tools could be used in a complementary manner leveraging their various capabilities on the same event records—saving repetitive steps when dealing with monolithic instrumentation chains. This would generalize what is already possible around the Score-P infrastructure.

3.2 The Producer/Consumer Interface

There are certain aspects of a unified interface that outline and motivate important design choices. First, if different tools choose to implement their own instrumentation chain, it is often because they intend to explore new kinds of source events and consequently cannot immediately use what is presently existing. However, with the rise of rich instrumentation interfaces (e.g., OMPT and MPI-T) and considering by itself the non-trivial task of instrumenting a parallel program, important efforts have to be directed to this relatively redundant work. The aim is to convert source events to a representation matching the analysis. instrumenting at event source from known interfaces and with common approaches (see Sect. 2.2). The interface, therefore, has to propose a highly-extensible event-model, both retro-compatible and future-proof. For example, if a new instrumentation interface rises, it should be possible to account for it without impacting tools. This first observation immediately forbids the use of the per-event function (as in OTF2), advocating instead for a single function taking an *event-object* as parameter. In this way, the consumer/producer interface can be fixed and this object later extended to match new requirements.

If we now consider this object representation in the context of a common language such a C, it will certainly be a `struct` pointer passed as a parameter through the interface. However, this struct should be able to represent an arbitrary number of events, each with their respective fields carrying metrics. In C, the `union` can be used to represent multiple data-layouts in the same object, this layout is chosen through the member name. As a consequence, it should be possible to determine the event "kind" which cannot be stored directly inside the union. Furthermore, any event occurring in a parallel application must be associated with meta-data, such as the thread ID where the event was triggered, the timestamp, the MPI rank, and so on. Like the event type, a "common" set of data has to be exposed for every event.

Figure 6 shows how a representation would allow multiple types to be encoded in the same struct without the constraint of the number of types supported thanks to the enum. However, the size of the C struct would be the header size plus the size of the largest enum member. Consequently, some "simple" events may have a footprint larger than they are. This would be a problem in the case of a trace format where storage size is crucial. This extra event footprint is then acceptable in comparison of the flexibility it provides when dealing solely with in-memory representations.

One aspect worthy of further consideration is meta-data. As shown in Fig. 6, events are contextualized to be associated with ranks and threads. However, the

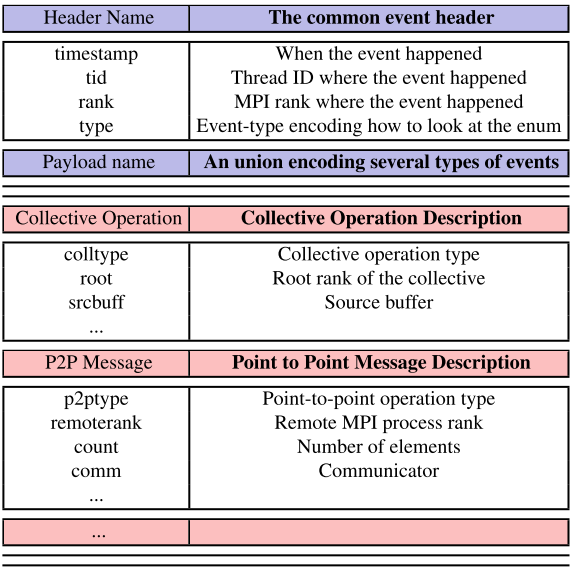


Fig. 6 Proposed data-model for an unified intermediate event representation

analyzer has to know of these threads and ranks and therefore must be informed of their existence. Most tools manage *shadow* contexts tracking all these identifiers to later retrieve representative state. Instead of this manually tracked state, the source events may simply emit an event for each new handle encountered (e.g., datatype, communicator, and so on). Thanks to logical event order for a given execution stream, the analyzer should be able to reestablish its *shadow* state based on the forwarded event, just as the dedicated instrumentation library would do when intercepting the events. This naturally outlines the need for dedicated meta-data forwarding events for new execution streams and handles and more specifically a common agreement on their semantics. This is probably the most difficult part in the design of a converging performance event meta-model as unlike for source-events, tools are free to choose an arbitrary state tracking approach.

Eventually, if we look at Fig. 5, the producer/consumer interface should be able to build a directed acyclic graph of tools. Producer interfaces should be exposed by name, allowing multiple tools (running in shared memory) to register and consume events. In particular, events should be repeated in each tool registered to the interface, for example, to both profile and trace in Fig. 5. Additionally, the producer interface should callback inside the consumer, as it simplifies data-parallelism which otherwise has to be manually implemented in the consumer. On this note, the single callback approach is very important to avoid the need to register several callbacks with a varying footprint to process events as in current OTF2 API. This idea is not new and has been explored in P^{n}MPI [22]. It has also been pursued in the generic tools interface (GTI) [14] and its transposition to tracing GTI-OTFX [24], together with TBON support. These related research efforts are examples of such modular and plugin-oriented infrastructures which adopt approaches close to the one we want to

motivate in this work. They clearly demonstrate the validity and interest of loading multiple components around an instrumented program. In fact, we see correlations in what is done in the GTI for validation/trace-analysis and MALP for profiling.

4 Practical Illustration with MALP

The Multi-Application Online Profiling (MALP) performance tool [5] has been designed around the concept of a `Generic Events` interface, matching the model we presented in Fig. 6. Our idea when we started the development was to find an alternative to verbose performance traces without sacrificing event verbosity too early in the instrumentation chain. In order to avoid I/O operations, we utilized an approach inspired from the P^{n}MPI [22] virtualization idea and forwarded a stream of events from the instrumented processes to the analysis. These events are similar to those you may find in a performance trace and rely on an intermediate representation. On the analysis side, a "blackboard" system allows plugins to register themselves to event types in order to perform data-reduction on incoming events. Eventually, reduced data are exported in a JSON file and presented in an HTML interface. In this description, one can see how MALP fits in the ideas we described in previous sections.

As presented in Fig. 7, we recently added support for new event-source interfaces such as MPI-T, Alinea MAP time-series, OTF2 trace format, and OMPT inside MALP. All events are then represented as `Generic Events` and forwarded over the network to be reduced inside analysis processes. Therefore, MALP is an illustration of the feasibility of the approach we proposed in this paper. It suggests that if we managed to find new inter-operability opportunities with existing tools we might be able to create bridges simplifying end-user experience. For instance, imagine that Score-P exposes a generic event interface. We would then be able to take advantage of

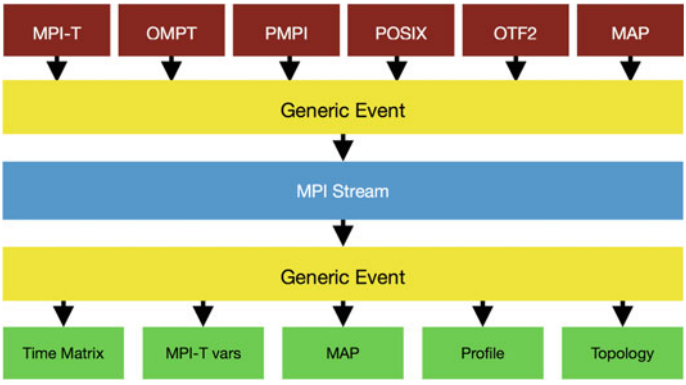


Fig. 7 Schematic overview of source-interfaces supported by the MALP performance tool

a state-of-the-art instrumentation substrate with a rich eco-system (packages, support, user-base), while enabling a robust connection to MALP for performance analysis. From a research and development point of view, it would have saved a lot of complexity in the tool implementation, saving redundant developments as instrumentation approaches are generally similar and well-known (see Sect. 2.2).

5 Conclusion

This paper proposes the concept of an intermediate event representation that that facilitates the development of performance tool workflows. Such workflows are based on a general instrumentation chain model, with components ranging from the instrumentation to the analysis. Our observation of transport-layer aspects of exemplar tools (e.g., post-mortem, in-place, offload, TBON) is that their data semantics is more or less invariant. Thus, we introduced the idea of modular instrumentation chains with an *intermediate representation* as mediation layer. In Sect. 3.1, we discussed the benefits that would arise from such data model, including new performance workflows and possible component reuse. We then suggested a C implementation using a struct of union and an unified handler interface, motivated by what we see as important design choices. Eventually, we discussed the implementation done in the MALP performance tools and showed how it related to our proposed model.

There is an interest in enabling tool inter-operability to simplify tools development and to provide more freedom to end-users who are currently forced to adopt the instrumentation chain for particular analysis tools. Related works such as Score-P, P^{n}MPI, the GTI, and various trace-converters show that there is a path for collaboration and modularity both inside and between tools. Our hope is that one day we will be able to compose performance tools workflows from a variety of components—TAU [23], Score-P, Paraver [21], HPCToolkit [1], Cube [11], MALP [5]. We believe that performance tools are currently not very far from being capable of this, in fact, a simple *in-memory* intermediate layer approaching the one we described in this paper for MALP would be sufficient. In the line of callback oriented instrumentation layers being developed for OpenMP and MPI, we are confident that infrastructures implementing this semantic will naturally overcome monolithic instrumentation chains.

6 Future Work

The MPI-T and OMPT source-events and simple analysis were implemented in MALP. A point which also needs attention is backtrace representation which requires dedicated storage and possibly complex in-place analysis to attach a given call-stack to events. We plan to explore the possibility of a unified space-efficient backtrace abstraction fitting in our proposed intermediate representation.

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
2. Ajima, Y., Inoue, T., Hiramoto, S., Uno, S., Sumimoto, S., Miura, K., Shida, N., Kawashima, T., Okamoto, T., Moriyama, O., Ikeda, Y., Tabata, T., Yoshikawa, T., Seki, K., Shimizu, T.: Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect, pp. 498–507. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07518-1_35
3. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: An Online-Based Distributed Performance Analysis Tool, pp. 1–16. Springer, Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_1
4. Besnard, J.B., Malony, A., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: An mpi halo-cell implementation for zero-copy abstraction. In: Proceedings of the 22Nd European MPI Users' Group Meeting, EuroMPI 2015, pp. 3:1–3:9. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2802658.2802669
5. Besnard, J.B., Pérache, M., Jalby, W.: Event streaming for online performance measurements reduction. In: 2013 42nd International Conference on Parallel Processing, pp. 985–994 (2013). https://doi.org/10.1109/ICPP.2013.117
6. Böhme, D., Gamblin, T., Beckingsale, D., Bremer, P., Giménez, A., LeGendre, M.P., Pearce, O., Schulz, M.: Caliper: performance introspection for HPC software stacks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016, pp. 550–560 (2016). https://doi.org/10.1109/SC.2016.46
7. Derradji, S., Palfer-Sollier, T., Panziera, J.P., Poudes, A., Atos, F.W.: The bxi interconnect architecture. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pp. 18–25 (2015). https://doi.org/10.1109/HOTI.2015.15
8. Eichenberger, A.E., Mellor-Crummey, J., Schulz, M., Wong, M., Copty, N., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis, pp. 171–185. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40698-0_13
9. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2: the next generation of scalable trace formats and support libraries. PARCO **22**, 481–490 (2011)
10. Fu, H., Liao, J., Yang, J., Wang, L., Song, Z., Huang, X., Yang, C., Xue, W., Liu, F., Qiao, F., Zhao, W., Yin, X., Hou, C., Zhang, C., Ge, W., Zhang, J., Wang, Y., Zhou, C., Yang, G.: The sunway taihulight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 072,001 (2016). https://doi.org/10.1007/s11432-016-5588-7
11. Geimer, M., Kuhlmann, B., Pulatova, F., Wolf, F., Wylie, B.J.N.: Scalable collation and presentation of call-path profile data with cube. In: Parallel Computing: Architectures, Algorithms and Applications: Proceedings Parallel Computing (ParCo07, Jlich/Aachen, pp. 645–652. IOS Press
12. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.1556
13. Giménez, A., Gamblin, T., Bhatele, A., Wood, C., Shoga, K., Marathe, A., Bremer, P.T., Hamann, B., Schulz, M.: Scrubjay: deriving knowledge from the disarray of hpc performance data. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 35:1–35:12. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3126908.3126935

14. Hilbrich, T., Müller, M.S., de Supinski, B.R., Schulz, M., Nagel, W.E.: Gti: a generic tools infrastructure for event-based tools in parallel systems. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 1364–1375 (2012). https://doi.org/10.1109/IPDPS.2012.123
15. Hilbrich, T., Schulz, M., Brunst, H., Protze, J., de Supinski, B.R., Müller, M.S.: Event-Action Mappings for Parallel Tools Infrastructures, pp. 43–54. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_4
16. Islam, T., Mohror, K., Schulz, M.: Exploring the capabilities of the new MPI_T interface. In: Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA 2014, pp. 91:91–91:96. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642769.2642781
17. de Kergommeaux, J.C., de Oliveira Stein, B.: Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions, pp. 133–140. Springer, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-44520-X_17
18. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir, pp. 79–91. Springer, Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-31476-6_7
19. Malony, A.D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., Dietrich, R., Poole, D., Lamb, C.: Parallel performance measurement of heterogeneous parallel systems with gpus. In: 2011 International Conference on Parallel Processing, pp. 176–185 (2011). https://doi.org/10.1109/ICPP.2011.71
20. Mohr, B., Malony, A.D., Shende, S., Wolf, F., et al.: Towards a performance tool interface for openmp: an approach based on directive rewriting. In: Proceedings of the Third Workshop on OpenMP (EWOMP01) (2001)
21. Pillet, V., Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., Girona, S., Computadors, D.D.D.: Paraver: a tool to visualize and analyze parallel code. Technical report, In WoTUG-18 (1995)
22. Schulz, M., de Supinski, B.R.: PNMPI tools: A whole lot greater than the sum of their parts. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, pp. 30:1–30:10. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1362622.1362663
23. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482
24. Wagner, M., Hilbrich, T., Brunst, H.: Online performance analysis: an event-based workflow design towards exascale. In: 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and System (HPCC,CSS,ICESS), pp. 839–846 (2014). https://doi.org/10.1109/HPCC.2014.145
25. Wolf, F., Mohr, B.: EARL—A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs, pp. 503–512. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/BFb0100611

OMPT-Multiplex: Nesting of OMPT Tools

Joachim Protze, Tim Cramer, Simon Convent and Matthias S. Müller

1 Introduction

In version 5.0 the OpenMP specification [1] will define a tool interface (OMPT) that allows monitoring tools to gain insights into implementation specific information about the execution behavior of an OpenMP application. The OMPT interface provides information about certain events during the execution, but also allows to query the OpenMP runtime about state and stack frame information. The interface is designed to enable the usage of a single OMPT tool. However, in order to enable a modular tool design, it might be helpful to split a single tool into parts which potentially need the same or distinct information from the OpenMP runtime. In other situations, it is desired to apply multiple tools at the same time. For MPI applications, this motivated the development of P^{n}MPI [2] which allows to use multiple PMPI tools at the same time.

In this paper we propose a method which allows a standard-compliant cascading of multiple OMPT tools.[1] The basic idea is that an OMPT tool can behave like the OpenMP runtime and provides the entire OMPT interface for the next tool. At the same time this paper describes the basic steps for creating a general OMPT tool.

[1] An implementation is available at https://git.rwth-aachen.de/OpenMPTools/OMPT-Multiplex.

J. Protze (✉) · T. Cramer · S. Convent · M. S. Müller
RWTH Aachen University, IT Center, Seffenter Weg 23, 52074 Aachen, Germany
e-mail: protze@itc.rwth-aachen.de

T. Cramer
e-mail: cramer@itc.rwth-aachen.de

S. Convent
e-mail: convent@itc.rwth-aachen.de

M. S. Müller
e-mail: mueller@itc.rwth-aachen.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_5

2 Use Cases

In this section, we discuss two different use cases, which represent common scenarios for using multiple OMPT tools at the same time. We conclude the section with an overview about tools that already adopt the OMPT interface.

2.1 Affinity Display

One of the new features introduced by OpenMP 5.0 will be a standardized way to report the affinity information by exporting the environment variable `OMP_DISPLAY_AFFINITY`. This enables the user to verify the intended affinity setting at execution time in a standard-compliant manner. During the discussion of this new feature, we already demonstrated that OMPT holds all necessary information to provide the requested output [3]. An OpenMP runtime implementation might use this OMPT based affinity display implementation to provide the specified behavior. This approach means, that the OpenMP runtime would load the affinity display tool instead of loading an OMPT tool following the work flow described in Sect. 4. Furthermore, this would occupy the OMPT interface and no other tool could be loaded at the same time. Working with tools has shown that they can break expected behavior. So it is important to have the ability to display the affinity information while a monitoring tool is active. Therefore, the affinity display tool needs to preserve the ability of loading another tool when active. Since the affinity tool in this scenario is directly loaded by the runtime skipping the OMPT tool initialization, the affinity tool needs to perform the full OMPT tool initialization on behalf of the OpenMP runtime.

2.2 Supplementary Debugging Tool

Besides OMPT, OpenMP 5.0 will also introduce a debugging interface (OMPD). This interface enables a third-party tool, i.e., a tool running in a separate process—potentially on a different machine—, to introspect and understand the state of the OpenMP runtime without any knowledge about the meaning of internal variables and structures. The defined OMPD library accepts high-level queries from the tool. The OMPD library then answers these queries by interpreting the memory in the OpenMP application. Since the OMPD library has the knowledge about the structure of the OpenMP runtime encoded, this library is closely coupled with a specific OpenMP runtime version.

If a debugger wants to provide additional state information that is not collected and provided by the OpenMP runtime and the OMPD library, an OMPT-based supplementary debugging tool can collect this information for the debugger. This infor-

mation can be related to task dependences, queued tasks, statistics about queued tasks, or stack information from task creation. This supplementary OMPT tool is closely paired with the debugging tool. This way, the debugging tool can understand the data layout in the OMPT tool and extract the collected information. In this case, the OMPD library supports the debugger in finding the right information by making the OMPT data pointer for the selected OpenMP scope accessible.

However, since OMPT natively only supports a single OMPT tool, this would prevent to use a monitoring tool while having the supplementary debugging tool in place. Therefore, this tool should also allow another tool to be loaded.

2.3 Other Tools Building on OMPT

Although the specification of OMPT is not finalized yet, several tools already were adapted to this new interface. Lorenz et al. [4] compare OPARI and OMPT in the context of the Score-P measurement infrastructure. HPCToolkit [5] is another performance analysis tool that took part in the co-design of OMPT and therefore adapted to the OMPT interface.

In the context of the data race detection tool Archer, we evaluated the synchronization information provided by OMPT [6]. Similar to the affinity tool, the OMPT tool provided by Archer is another candidate for a tool that finally might be tightly integrated into the compiler and gets automatically activated in the case that a special compiler feature is active. This is only possible if the tool does not block the OMPT interface.

3 OMPT Multiplex Architecture

In order to overcome the limitation of a single OMPT tool, we developed a multiplex tool. This multiplex tool is provided as a header file, that can be compiled into any OMPT tool. The header enables the including tool to load another OMPT tool. Since the multiplexing tool is header-only, there is no separate library to load. In the following we will refer the *including* tool as the *own tool* and the *nested* tool as the *client tool*.

Figure 1 shows the basic steps and concepts of any OMPT tool:

1. Runtime initialization:
   - The OpenMP runtime searches for an OMPT tool and calls **`ompt_start_tool`**
   - After the OpenMP runtime finished initialization, it allows the tool to initialize by calling **`ompt_initialize`**

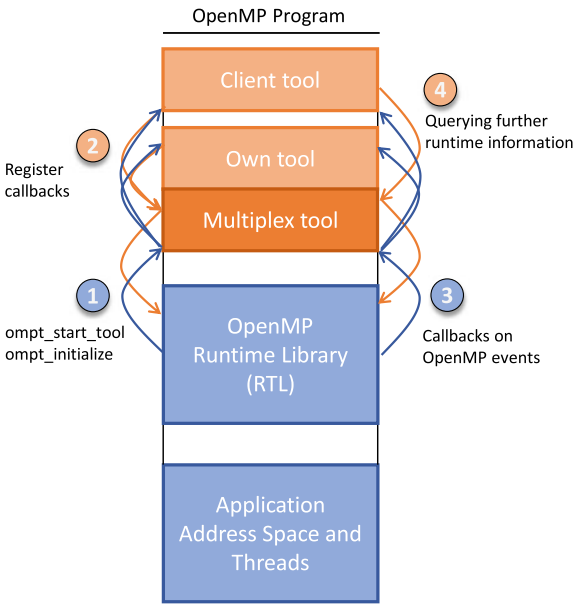


Fig. 1 OMPT-Multiplex Tool architecture: The multiplex tool is built into the *own tool* and loads a *client tool*. All OMPT API function calls are redirected to the multiplex tool

2. Tool initialization:
   - A tool looks up OMPT functions with the **`lookup`** function provided as argument of the **`ompt_initialize`** function
   - A tool registers callbacks for OpenMP events using **`ompt_set_callback`**

3. Delivery of OpenMP events:
   - The OpenMP runtime invokes tool callbacks during the execution of the program whenever an event is encountered (e.g., entering/exiting parallel, target, worksharing, or synchronization regions)

4. Querying further information from the OpenMP runtime:
   - A tool might call OMPT functions to investigate the thread state or identify OpenMP runtime frames

For the multiplex tool, these steps have the following additional implications:

1. Runtime initialization (see Sect. 4):
   - The multiplexing tool collects the initialize and finalize function pointers for the *own tool* and the *client tool*.
   - For the initialize and finalize function call from the runtime, the multiplexing tool calls the collected function of the *own tool* and the *client tool*.

2. Tool initialization (see Sect. 6):
   - The tools register their callbacks with the multiplex tool.
   - The multiplex tool registers callbacks with the runtime if either one of the tools registered a callback for the event.
3. Delivery of OpenMP events (see Sect. 8):
   - For each callback invocation, the multiplexing tool first calls the registered function of the *own tool*, then of the *client tool*.
   - The multiplexing tool provides tool specific data as discussed in Sect. 5.
   - Exceptions for the ordering of callback invocations are discussed in Sect. 8.
4. Querying further information from the OpenMP runtime (see Sect. 7):
   - Some of the OMPT functions are replaced by the multiplexing tool to allow multiplexing of the tool specific data pointer.

4 Activating an OMPT Tool

According to the OpenMP specification, an OpenMP runtime implementation follows several steps in trying to find the function **`ompt_start_tool`**. If calling this function returns **`true`**, the runtime has found an OMPT tool, otherwise the runtime would continue finding a tool. The OpenMP specification describes the following methods how the function can be provided: It can be

- statically linked into the application,
- dynamically linked into the application (this includes loading the library with **`LD_PRELOAD`**) or
- provided by a dynamically-linked library listed in the **`OMP_TOOL_LIBRARIES`** environment variable.

For our cascading approach we limit the search space to the latter option. Each tool selects and documents a name for searching the *client tool*. When including the **`OMPT-Multiplex.h`** header, this name must be defined in the preprocessor define **`CLIENT_TOOL_LIBRARIES_VAR`**::

```
#define CLIENT_TOOL_LIBRARIES_VAR
  ↪   "<EXAMPLE>_TOOL_LIBRARIES"
#include <ompt_multiplex.h>
```

This header file must be included in the tool source file that implements **`ompt_start_tool`**. The header renames the tool's **`ompt_start_tool`** function to overwrite this function with the cascading implementation of this function. The cascading **`ompt_start_tool`** function invokes the renamed function of the tool and then tries to find the *client tool* in the libraries listed in the **`<EXAMPLE>_TOOL_LIBRARIES`** environmental variable.

5 Tool Data Pointer

In multiple occasions, the OMPT interface allows the tool to store information into a tool data value. This value is defined to be a union of a void pointer and a 64 bit integer:

```
typedef union ompt_data_t
  {uint64_t value; void * ptr;} ompt_data_t;
```

An OMPT tool can store a single value directly in this `ompt_data_t` value. If a tool wants to bind more information to an OpenMP entity, the tool would allocate memory for this information and store the address in this `ompt_data_t` value.

Since both, the *own* and the *client tool*, expect to store their private data into this value, the cascading tool needs to multiplex the data correspondingly. The header file provides two modes for the organization of the own and the client data. In the *simple mode* no further modification of the *own tool* is necessary. In the *advanced mode* the *own tool* is aware of the cascading tool.

Figure 2 depicts the simple mode, where both tools are unaware of the other tool. The multiplexing header allocates a pair of `ompt_data_t` values and stores the address of this pair in the runtime. This pair is used to store the `ompt_data_t` value of the *own* and the *client tool*. This mechanism applies to all callbacks which have a `ompt_data_t` in their signature, i.e. data pointer for thread, parallel region, and task, as well as the data pointer in the return value of the `ompt_start_tool` function.

For some of these tool data objects, the life cycle might be quite short. To avoid the overhead of the additional allocation, it is also possible that an aware tool can provide storage for the *client tool* data object. This advanced mode is depicted in Figure 3. For tools that store objects with tool-specific data ("OwnData") in the scope specific data pointer, it is easy to extend the object by an `ompt_data_t` field to store the client data. *Scopes* as known from the OpenMP specification are device, thread, parallel team, and task. The multiplexing header needs to know how to access

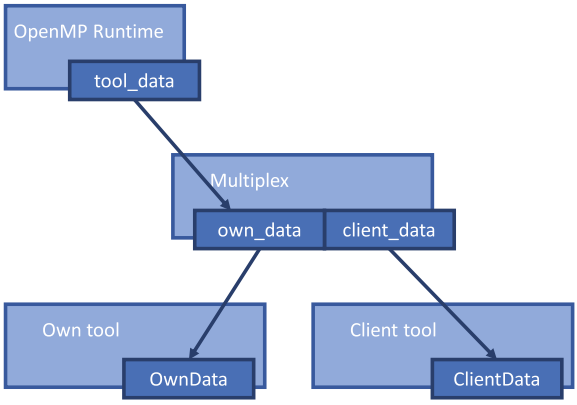


Fig. 2 Simple tool data management: Multiplex-header allocates memory to store a pair of tool data for *own* and *client tool*

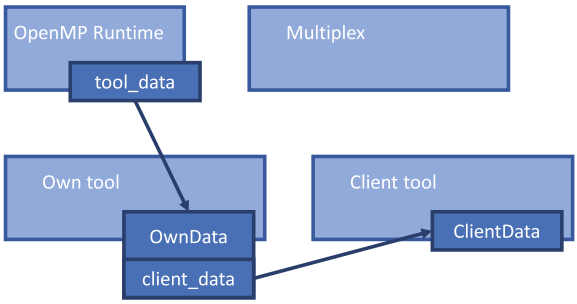


Fig. 3 Advanced tool data management: Own tool provides memory to store tool data for the *client tool*

this data. Therefore, the tool defines an accessor function for the data field of the prototype:

```
ompt_data_t * get_client_data (ompt_data_t* data);
```

The name of the accessor function is propagated to the multiplex header by the following defines:

- `OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_THREAD_DATA`
- `OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_PARALLEL_DATA`
- `OMPT_MULTIPLEX_CUSTOM_GET_CLIENT_TASK_DATA`

If the accessor macro for a scope specific data pointer is defined, the multiplexing header passes the pointer from the runtime directly to the *own tool* and calls the accessor function to pass the data pointer to the *client tool*. Otherwise the multiplexing header uses the allocated pair and provides the first value to the *own tool* and the second value to the *client tool*. If a tool does not use the tool data for a specific scope, this macro can be an empty definition, so that the runtime data pointer is directly passed to the client tool.

6 Initializing an OMPT Tool

When the OpenMP runtime finished initialization of internal data structures and is ready to accept function calls from the OMPT tool, the runtime invokes the initialization function that was provided by the `ompt_start_tool` function (see Sect. 4):

```
int ompt_initialize (ompt_function_lookup_t lookup,
 ↪  ompt_data_t* data);
```

The first argument of the initialization function is a lookup function, which is used by the OMPT tool, to find the OMPT runtime entry points. The second argument of the initialization function is a pointer to tool specific data. This data was passed by

the tool on return from the `ompt_start_tool` function. The multiplexing tool makes sure to pass back the right pointers to the *own tool* and the *client tool*.

The multiplexing tool implements two versions of this lookup function, one for the *own tool* and one for the *client tool* and passes the pointers to these functions accordingly. For all OMPT functions—the OpenMP document calls them OMPT runtime entry points—with an `ompt_data_t` argument the lookup function provides a customized function; these functions implement the multiplexing of the tool data pointer. For all other runtime entry points, the lookup function just forwards the query to the runtime; these function calls have no additional indirection.

Besides looking up the runtime entry points, a tool would typically register some callback using the `ompt_set_callback` function. The multiplexing tool itself doesn't need any callbacks set; the setting of callbacks is delegated to the two customized `ompt_set_callback` functions for the *own tool* and the *client tool*. When either of the two tools registers for a callback, the multiplexing tool registers the own corresponding callback.

7 Runtime Entry Points

The OMPT API functions provided by the OpenMP runtime are called runtime entry points. The only way to get access to these functions is the lookup function provided in the initialization call. The runtime does not need to implement these functions under these names, a tool cannot directly call these functions. The advantage is that a tool does not link against an OpenMP runtime. This allows more flexibility for the tool, since the same tool can be used with OpenMP and non-OpenMP applications.

Furthermore, this restriction provides the guarantee to the runtime, that these functions can only be called after the runtime is initialized; this differs from the OpenMP runtime routines (e.g., `omp_get_num_threads()`), where the runtime needs to check whether initialization is already done.

As mentioned in the previous section, there is only a small number of runtime entry points that need special care from the multiplexing tool. For these functions, the *own tool* and the *client tool* expect to access their own tool specific data; so the multiplexing tool needs to know which data should be passed to the tool. The OMPT interface does not allow to pass additional information with the function call. Consequently, the only way to know whether a call to a runtime entry point comes from the *own tool* or the *client tool* is by providing them distinct functions for the same entry point. The provided function implicitly knows whether it should access the *own tool* or the *client* data field.

8 Callback Functions

An OMPT tool can register callback functions for several defined OpenMP events. If the execution of the OpenMP program reaches an OpenMP event, the runtime invokes the registered OMPT callback function for this event. The multiplexing tool keeps track which tool registered a callback for each event. If both tools registered a callback for the same event, in general the callback of the *own tool* is called first and then the callback of the *client tool*.

In the advanced mode where the *own tool* provides storage for the tool data of the *client tool*, this ordering might fail. For the parallel-end or the implicit-end event, the *own tool* would release the data structure. The multiplexing tool would then access the data pointer in the already freed data structure to pass it in the callback to the *client tool*.

The multiplexing tool provides two different strategies for circumventing this issue when the *own tool* provides storage for the *client* data:

1. For the callbacks that describe the end of a data environment, the multiplexing tool invokes the callback of the *client tool* first.
2. The original ordering is preserved, if the *own tool* implements a delete function (e.g., **`delete_data()`**) for the data object and registers this function by defining scope specific macros before including the multiplexing header:

```
void delete_data(ompt_data_t*);
#define OMPT_MULTIPLEX_CUSTOM_DELETE_<SCOPE>_DATA
  ↪  delete_data
```

The first strategy allows the *client tool* to release the scope specific tool data before the *own tool* releases the tool data and deletes the pointer to the *client* tool data. In the second strategy, the multiplexing tool calls the **`delete_data()`** function after invoking the callback of the *own tool* and the *client tool*. This way we can safely preserve the ordering. This assumes that the callback of the *own tool* does not destruct the scope specific data object, but leaves this task to the delete function.

9 OMPT-Multiplex Versus OMPT^n

The initial idea when designing an infrastructure to use multiple OMPT tools at the same time was to use an approach similar to P^{n}MPI. P^{n}MPI is a tool for the PMPI interface of MPI, that allows multiple PMPI tools to be loaded during execution. The PMPI interface is different from OMPT, as a tool intercepts the MPI function calls and then calls the PMPI function to invoke the actual function in the MPI runtime library. P^{n}MPI then redirects the function call to the next tool before finally the MPI runtime library is called.

Based on this model, OMPT^n would be a stand-alone OMPT multiplexing tool which loads multiple OMPT tools, distributes callback functions to all loaded tools

and multiplexes all the tool specific data. The problem we are facing with this approach is the function pointer passed in the lookup function. The function pointer needs to be a C function pointer. There is no way to pass additional information to the tool for the lookup function. With additional information passed, it would be possible to implement the scope specific functions in a way, that they multiplex the tool specific data and deliver the right data to the right tool. A modern C++ approach of implementing this dynamic multiplexing might be the use of lambda expressions. With a capturing lambda expression, the information about the tool might be coded into the function object. The problem is again, that no plain C function pointer can be derived from a capturing lambda expression, because the additional information cannot be encoded. The only way to implicitly encode the information about the tool instance into the OMPT runtime entry point is to statically provide versions of the function that encode the access to the right tool data. This would limit the maximum number of tools that can be used at the same time.

10 Recursive Use of OMPT Multiplex

For the approach with the OMPT-Multiplex header, there is no limitation in the maximum number of tools that can be nested. As long as each tool includes the header, the number of tools that can be nested recursively is only limited by the stack size.

The remaining limitation of this approach is that each tool library is compiled with a hard coded name for the environmental variable. Therefore the same tool library cannot be loaded twice since this would lead to an infinite recursion. So, if for some reason the same tool needs to be loaded multiple times, multiple versions of the library are necessary that are compiled with different names for the tool libraries variable. Also two different tools cannot use the same tool libraries variable.

For tools that use an execution wrapper, inserting the tool into the chain can happen transparently to the user. The execution wrapper assigns the content of **`OMPT_TOOL_LIBRARIES`** to the tool libraries variable of the own tool and updates the **`OMPT_TOOL_LIBRARIES`** to point to the own tool library.

11 Conclusion

In this paper we proposed a method to enable the usage of multiple OMPT tools at the same time. We provide a freely available implementation of this method in form of a header-file-only solution, which is simply included by the OMPT tool. A requirement is that the header file is included by the tool at compilation. For proprietary tools that are delivered only as precompiled binary, the vendor would need to use this header to

enable the proposed workflow. With this work we also proved that the specification of OMPT allows the usage of multiple OMPT tools at the same time, although the specification only supports a single tool.

References

1. OpenMP Architecture Review Board: TR4: OpenMP Version 5.0 Preview 2. http://www.openmp.org/wp-content/uploads/openmp-tr6.pdf
2. Schulz, M., de Supinski, B.R.: PNMPI tools: a whole lot greater than the sum of their parts. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing. SC 2007, pp. 1–10 (2007)
3. OpenMP Affinity Tool. https://git.rwth-aachen.de/OpenMPTools/OMPT-Affinity-tool
4. Lorenz, D., Dietrich, R., Tschüter, R., Wolf, F.: A comparison between OPARI2 and the OpenMP tools interface in the context of Score-P. In: Proceedings of the 10th International Workshop on OpenMP (IWOMP), Salvador, Brazil, September 2014. LNCS, vol. 8766, pp. 161–172. Springer International Publishing (2014)
5. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: tools for performance analysis of optimized parallel programs. In: Concurrency and Computation: Practice and Experience, pp. 685–701 (2010)
6. Protze, J., Hahnfeld, J., Ahn, D.H., Schulz, M., Müller, M.S.: OpenMP tools interface: synchronization information for data race detection. In: Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20–22, 2017, Proceedings, pp. 249–265 (2017)

SCIPHI Score-P and Cube Extensions for Intel Phi

Marc Schlütter, Christian Feld, Pavel Saviankou, Michael Knobloch, Marc-André Hermanns and Bernd Mohr

Abstract The Intel®Xeon Phi™ Knights Landing processors offers unique features with regards to memory hierarchy and vectorization capabilities. To improve tool support within these two areas, we present extensions to the Score-P measurement infrastructure and the Cube report explorer. With the Knights Landing edition, Intel introduced a new memory architecture, utilizing two types of memory, MCDRAM and DDR4 SDRAM. To assist the user in the decision where to place data structures, we introduce a MCDRAM candidate metric to the Cube report explorer. In addition we track all MCDRAM allocations through the hbwmalloc interface, providing memory metrics like leaked memory or the high-water mark on a per-region basis, as already known for the ubiquitous malloc/free. A Score-P metric plugin that records memory statistics via numastat on a per process level enables a timeline analysis using the Vampir toolset. To get the best performance out of Intel®Xeon Phi™, the large vector processing units need to be utilized effectively. The ratio between computation and data access and the vector processing unit (VPU) intensity are introduced as metrics to identify vectorization candidates on a per-region basis. The Portable Hardware Locality (hwloc) Broquedis et al. (hwloc: a generic framework

M. Schlütter (✉) · C. Feld · P. Saviankou · M. Knobloch
Forschungszentrum Jülich GmbH JSC, Jülich Supercomputing Centre,
Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
e-mail: m.schluetter@fz-juelich.de

C. Feld
e-mail: c.feld@fz-juelich.de

P. Saviankou
e-mail: p.saviankou@fz-juelich.de

M. Knobloch
e-mail: m.knobloch@fz-juelich.de

M.-A. Hermanns · B. Mohr
JARA-HPC, Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH,
Jülich, Germany
e-mail: m.a.hermanns@fz-juelich.de

B. Mohr
e-mail: b.mohr@fz-juelich.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_6

for managing hardware affinities in hpc applications, 2010 [2]) library allows us to visualize the distribution of the KNL-specific performance metrics within the Cube report explorer, taking the hardware topology consisting of processor tiles and cores into account.

1 Introduction

Many-core architectures like the Intel®Xeon Phi™ provide opportunities and challenges for intra-node optimization of applications. The Intel®Xeon Phi™ Knights Landing (KNL) comes with a unique memory hierarchy and a 512-bit-wise vector processing unit. To gain the full benefits of the new features, the user needs to understand how effectively an application makes use of the underlying hardware capabilities. The objective of the SCIPHI (Score-P and Cube extensions for Intel Phi) project has been to incorporate this knowledge in the Score-P and Cube tools and provide it to their users.

Figure 1 shows a schematic image of the KNL chip. It highlights some areas of interest for the SCIPHI project, specifically the memory topology and tiled hardware layout.

A KNL chip consists of 38 uniform tiles [15], of which at most 36 are enabled. Each tile comes with two Silvermont cores—supporting up to four hyperthreads—and two Advanced Vector Extensions, known as AVX-512 [5] vector processing units (VPU). The high number of VPUs and the lower clock frequency of the Silvermont cores, compared to recent Xeon CPUs, makes vectorization not just an opportunity, it becomes a necessity for efficient node usage.

Besides the AVX-512 vector units, KNL is special with regards to memory. It comes with two types of memory, conventional DDR4 SDRAM providing high capacity and MCDRAM (High-Bandwidth Memory) providing high bandwidth. There are eight MCDRAM modules integrated on package, providing a total of 16 GB high bandwidth memory. These devices come with their own memory controller, providing a total bandwidth of more than 450 GB/s (Stream Triad [15]). The DDR4 SDRAM memory, on the other hand, is connected via two controllers, serving three channels each. The maximum capacity is 384 GB and the bandwidth can reach up to 90 GB/s.

The memory can be configured at boot time in one of three modes. In *cache mode*, MCDRAM serves as a cache for DDR4 SDRAM. In *flat mode*, MCDRAM is treated as standard memory in the same address space as DDR4 SDRAM; the *memkind* library[1]—a heap manager built on top of *jemalloc*[2]—allows for MCDRAM heap allocations via the *hbwmalloc* API. *Hybrid mode* is a combination of cache and flat mode, where a portion of MCDRAM serves as cache while the remainder can be

[1] https://github.com/memkind.

[2] http://jemalloc.net/.

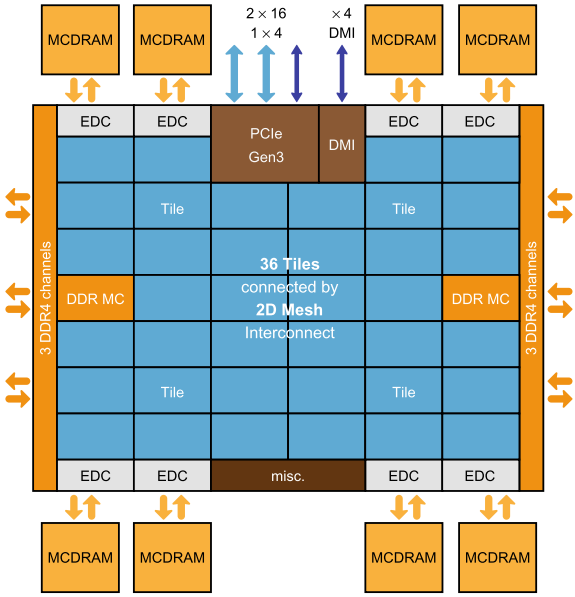


Fig. 1 Intel®Xeon Phi™ Knights Landing architecture [15]

use a standard memory. Cache friendly applications are likely to benefit from *cache* mode. There are applications though that might benefit from explicit MCDRAM memory management in *flat* mode.

In this paper we present extensions to the scalable performance measurement infrastructure for parallel codes *Score-P* [9] and the performance report explorer *Cube* [16] with regard to the special memory and vectorization features of the Intel®Xeon Phi™ Knights Landing processor. In particular, we implement means to track memory allocations and deallocations for both, DDR4 SDRAM and MCDRAM memory. In addition, a MCDRAM candidate metric on a per region basis is introduced. With respect to vectorization, we present VPU-related metrics that point the user to code regions that would benefit from vectorization. In addition we utilize the *hwloc* [2] library to visualize the distribution of the KNL-specific performance metrics, taking the hardware topology into account.

The paper is structured as follows. Section 2 contains a survey of related work. Sections 3 and 4 focus on the two main topics, memory analysis and vectorization support. Section 5 presents the changes and requirements for the Score-P measurement work-flow that result from the previous sections. In Sect. 6 we present the Score-P hardware topology visualization in Cube using the example of the KNL architecture. We close with a conclusion and an outlook of future extensions in Sect. 7.

2 Related Work

Investigating memory usage, performance analysis tools, such as Score-P [9], TAU [18], and Vampirtrace [8], follow their call-path oriented approach in assigning measurement data to code regions. Jurenz et al. [7] highlight different methods for querying memory data and how they are used in Vampirtrace. In contrast, tools, such as Intel VTune [6] or HPCToolkit [1], provide a data-centric perspective. For example, Liu et al. [10] present an extension to the HPCToolkit, using instruction-based sampling to record memory usage through relevant events, pinpointing to an effective memory address.

Vectorization and memory related metrics use hardware counters as basis for calculation, accessed either directly or through third party software. PAPI [3] provides such access to hardware counters, in the KNL case also to the node level uncore counters, which are required for the analysis of memory accesses and bandwidth. As an alternative, the LIKWID tools [19] allow similar use of counter information on the KNL, including access to shared counters. Wylie et al. [21] highlights that depending on the type measuring multiple hardware counters at the same time can be an issue and proposes a solution through multiple manual measurements. Reinders et al. [15] suggest metrics and thresholds for the KNL architecture, which should serve as guidelines for the user during optimization.

Scalasca version 1.x [12], still using its internal measurement system, provided topologies for Cube, while such support in Score-P is scheduled for a future release.

3 Memory Analysis

The availability of high bandwidth MCDRAM on the Intel®Xeon Phi™ Knights Landing provides unique opportunities as well as challenges for the application developer. The potential increase in bandwidth when using MCDRAM is counterbalanced by the reduced capacity available. With only 16 GB MCDRAM, compared to the maximum of 384 GB DDR4 SDRAM, the use of MCDRAM has to be managed carefully. If the KNL is booted in *cache mode*, the developer has no direct access to the MCDRAM, but the system uses this up to 16 GB as an additional, transparent cache for DDR4 SDRAM data. However, if the node is booted in *flat mode*—and to a degree in *hybrid mode*—the developer is responsible to explicitly manage allocations from MCDRAM. In these modes, at least one additional NUMA node is present, depending on the selected cluster mode. Allocations and deallocations can be managed in several ways. One way is to utilize the *NUMA Control utility* (numactl). This method requires no code changes but the allocations must completely fit into MCDRAM, as all memory is allocated from this NUMA node, including the data segments and stack. Another method without the need of code changes is the use of *autohbw*, which comes with the memkind package. This library allocates memory chunks of a certain size range transparently from MCDRAM. Yet

another way to handle allocations into MCDRAM is provided by the *hbwmalloc* API that also comes with the *memkind* library. This API offers replacements for glibc's malloc routines in C and C++ and the FASTMEM directive for Intel Fortran. This method provides the maximum amount of control but comes at the cost of mandatory code changes.

Given these three methods to explicitly manage MCDRAM and the limited amount of high bandwidth memory, the developer is faced with the question of *what* and *how much*, if not all, to allocate from MCDRAM. By providing relevant information about memory usage, tools can support the user in the decision-making process.

3.1 DDR4 SDRAM and MCDRAM Usage

If the working set fits into the 16 GB of MCDRAM, the easiest way to utilize the high bandwidth memory is to use numactl. The command `numactl -H` gives us the available NUMA nodes and with `numactl -m 1 ./application` we start a program that allocates all memory from NUMA node 1, which maps to MCDRAM in flat/quadrant mode.

If the working set is larger than the available 16 GB, shifting all memory allocations to MCDRAM will fail. To determine the actual memory requirements of an application, one can observe fine-grained allocations by tracking calls to `malloc`, `free`, and similar functions.[3] A more course-grained approach is to monitor the output of `numactl`, `numastat`, or `getrusage`. The open-source measurement infrastructure Score-P [9] already provides the functionality to observe the fine-grained allocations by wrapping the necessary library calls. This allows to determine allocations and deallocations on a per-region and per-thread basis. It also points the developer to leaked memory, i.e., it shows allocations that haven't been deallocated. Last but not least it keeps track of the used memory's *high-water mark*, i.e., the maximum amount of memory allocated at a time and points the developer to the code region where this maximum was reached. These recorded memory metrics can be analyzed in CUBE's visualization of call-path profiles [16] and be timeline-visualized by the Vampir trace analyzer [8], and potentially any other tools that support the open Score-P output formats *cubex* and *OTF2* [4].

The memory allocation tracking works only for conventional dynamic memory allocations, i.e., allocations from DDR4 SDRAM, as `malloc` and friends don't allocate from MCDRAM. By observing these conventional allocations one can estimate the memory requirements of an application and determine if the entire working set would fit into the 16 GB MCDRAM. The recorded allocations also provide the entire set of candidates that might benefit from being moved to MCDRAM.

[3] `malloc,realloc,calloc,free,memalign,posix_memalign,valloc`.

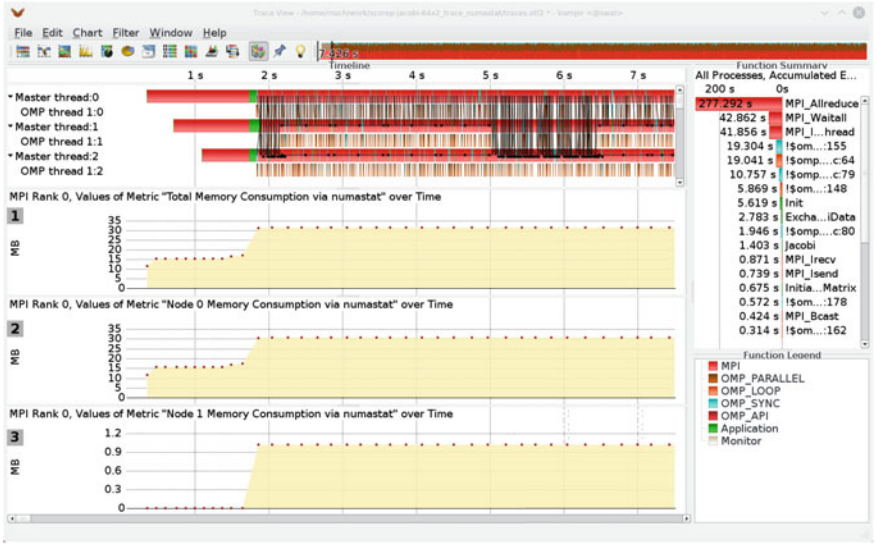
To also observe allocations, find leaks, and the high-water mark when explicitly working with MCDRAM, it seemed natural to extend Score-P to intercept the *hbwmalloc* API. In flat mode, this API allocates from MCDRAM and can be used as a drop-in replacement for the usual `malloc/free` set of functions. To track the hbwmalloc allocations, we added the following functions to the set of wrapped library functions, this way providing the same analysis opportunities as for conventional allocations: `hbw_malloc`, `hbw_calloc`, `hbw_realloc`, `hbw_free`, `hbw_posix_memalign`, and `hbw_posix_memalign_psize`.

Tracking allocations and deallocations in such a fine-grained way can induce significant measurement overhead with regards to the Score-P-internal memory requirements to perform the tracking, in particular in cases with excessive numbers of heap memory operations. Furthermore, a high-water mark of memory consumption below 16 GB does not guarantee that the application will fit entirely into MCDRAM. This is due to the fact that memory is allocated on a per-page basis (usually 4 K) on first touch. The first touch may happen way later in time than the `malloc` call and freed memory may not get reused. The actual memory consumption might be larger than reported by just tracking memory allocation and deallocation routines.

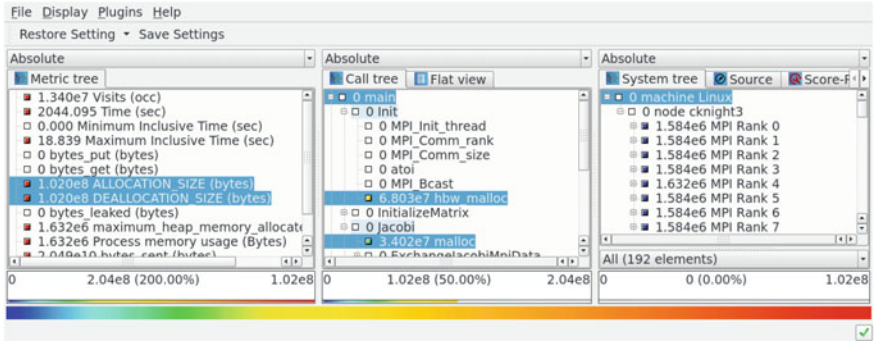
To get a more accurate measure of the used memory on a per-page basis we use the `numastat` command as a complimentary data source. It provides per-NUMA-node memory statistics for processes and the operating system. This allows not only for monitoring conventional memory allocations, but also for MCDRAM allocations in arbitrary cluster modes. Depending on the cluster mode—quadrant, SNC-2, or SNC-4, selected at boot time—a KNL chip in flat memory mode reports two, four, or eight NUMA nodes per quadrant, equally distributed among DDR4 SDRAM and MCDRAM. To monitor the evolution in time of these NUMA nodes with regards to memory, we implemented a Score-P metric plugin. Score-P metric plugins are (third-party) shared libraries that are linked to the measurement core in order to provide any kind of additional metrics. They are modeled after VampirTrace's [8] Counter Plugins [17]. Our numastat plugin is executd asynchronously, i.e., it is triggered at regular intervals within an extra thread, analyses the numastat output for the current process (`numastat -p <PID>`) and feeds the per-NUMA-node memory statistics into a Score-P timeline.

Figure 2 shows the visualization of such a timeline, generated by a simple Jacobi solver, using the Vampir tool set. For this demonstration some memory requests have been explicitly allocated from MCDRAM, using the hbwmalloc API. We can observe the DDR4 SDRAM and MCDRAM allocations in Vampir via an additional metric timelines alongside the master timeline. In our example, the numastat plugin records three additional timelines, the memory usage for each of the two NUMA nodes and the total memory usage. Here "Node 0"(2) corresponds to DDR4 SDRAM and "Node 1"(3) to MCDRAM.

The polling frequency of the numastat plugin can be chosen via an environment variable. If we are solely interested in the memory evolution over time we might accept the overhead introduced by a high polling frequency. But even with a low polling frequency in relation to the application run time the memory usage should be attributable to specifc phases of the application.



(a) Vampir Timeline



(b) Cube Profile

Fig. 2 Memory consumption over time as reported by numastat, separated into DDR4 SDRAM (Node 0), MCDRAM (Node 1), and Total in the Vampir timeline, alongside with a corresponding Cube report with activated hbwmalloc tracking

3.2 MCDRAM Candidates

If the working set does not fit into the 16 GB of MCDRAM it might be beneficial to allocate parts of the working set datastructures into high bandwidth memory using the hbwmalloc API. But such a change in allocation does not always improve the runtime. For the CloaverLeaf [13] hydrodynamics mini-app, e.g., we see runtime reduction due to selective allocations into MCDRAM as compared to pure conventional allocations only for high overall DDR4 SDRAM bandwidth values, see Fig. 3. To create this

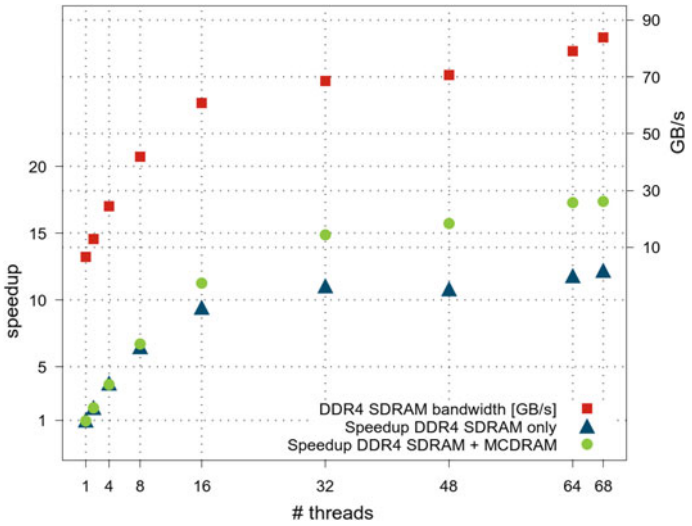


Fig. 3 CloverLeaf3D on a single KNL. Baseline for speedup is DDR4 SDRAM-only (blue) on a single thread. For DDR4 SDRAM + MCDRAM (green), some data structures were manually allocated into MCDRAM. DDR4 SDRAM bandwidth (red) is for the entire application, not individual kernels

graph, we manually changed the allocation for bandwidth sensitve datastructures using the `FASTMEM` directive according to the manually found optimum in [15]. We see improvements over the DDR4 SDRAM-only variant for higher thread counts that correspond to higher overall bandwidth. To obtain the bandwidth, we counted the number of read and write accesses to DDR4 SDRAM (`UNC_M_CAS_COUNT.ALL`) with the help of PAPI [3], multiplied by the cache-line size and divided over time. It is important for the process to have exclusive acccess to the KNL-node as obtaining the memory accesses is done via the uncore counters, which provide data for the entire node only, not for individual processes or threads. Hence, this analysis does not work, if the KNL-nodes is shared among jobs or users.

With Score-P, we measure the bandwidth values per code-region outside of OpenMP parallel regions, due the given uncore counter restrictions. Depending on the application, there might be a lot of code regions that show a *high* bandwidth value. To find the most bandwidth sensitive candidates among these regions, we need to sort them by their last-level cache-misses (LLC). This gives us the *MCDRAM candidate metric* per code region, as shown in Fig. 4. We derive the MCDRAM candidate metric, i.e., we sort the high bandwith callpaths by their last-level cache misses, in the Cube plugin *KNL advisor* (see also Sect. 5.2). As input we use the PAPI-measured access counts for each DDR4 memory channel and the PAPI-measured LLC counts. We take care of measuring the memory accesses only per-process while running exclusively on a single KNL node.

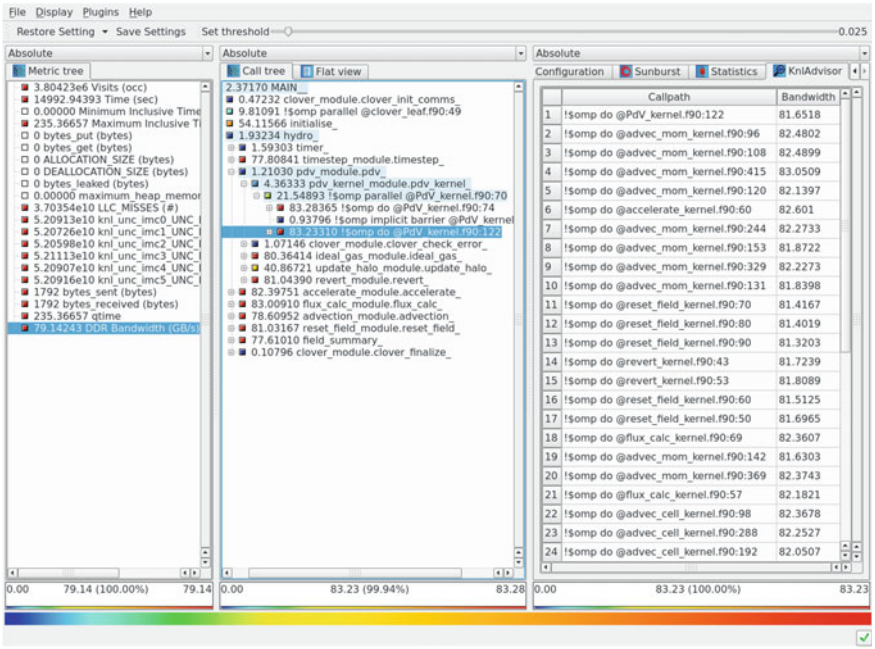


Fig. 4 Candidates for allocations in HBM: Highest bandwidth regions are sorted by last level cache misses

As Score-P and Cube purely work on code regions, the MCDRAM candidates are also code regions. As a drawback, if a candidate code region accesses several data structures, we cannot point to the most bandwidth sensitive structure. Vtune [6], HPCToolkit [1, 10] or ScaAnalyzer [11] might provide more detailed insight.

In addition to this drawback, the above approach is not generally applicable for tools as accessing counters from the uncore requires priviledged access to a machine, either by setting the paranoia flag or by providing a special kernel module. On production machines, this access is, for security reasons, often not granted. This does not only apply to memory accesses, but to all uncore counters.

4 Vectorization Assistance

In this section we focus on user support for vectorization, the second extension area of the SCIPHI project. Specifically, we investigate loops with regard to their degree of vectorization and offer suggestions for optimization candidates. This required hardware counter measurements, obtained in multiple runs, due to the limited number of available counter registers. In the context of counter measurements this is not unusual for the Score-P work-flow. The suggestion of specific optimization candidates on the

other hand is a deviation from the standard Score-P metric semantics. The Score-P metric concept operates on the actual value of a metric (in absolute or relative terms) and analysis sometimes requires implicit information, e.g. if a higher value is worse than a small value. This approach leaves the decision about the relevance of a metric value of a certain call-path to the user. They need to judge the severity of an issue based on the knowledge of the hardware architecture, the source code, the input data, the use case, or even external parameters. Providing a generic set of thresholds, deciding if a metric value is problematic, is a hard problem in general, as too many parameters are involved, some outside the scope of the performance analysis tool. In the case of vectorization assistance we used the cooperation with Intel® to investigate the use of explicit knowledge about the architecture for providing such thresholds in that limited context. In the following we describe the metrics we focused on and the challenges they pose for the Score-P work-flow and analysis.

4.1 Metrics

Before we can generate the candidate lists for vectorization optimization in Sect. 5.2, we need to define the metrics that form the basis for the selection. We focus on the three metrics, that are listed in Table 1. The first metric calculates the computational density, i.e. the number of operations performed on average for each piece of loaded data. The `L1 compute to data access ratio` can be used to judge how suitable an application is to run on the KNL architecture. Ideally, operations should be vectorized and each datum fetched from L1 cache should be used for multiple operations. Table 1 shows the formula as number of vector operations vs. the number of loads seen by the L1 cache. Similar to this, the `L2 compute to data access ratio` is calculated as the number of vector operations against the loads that initially miss the L1 cache. While the L1 metric is critical in estimating a codes general suitability, the L2 metric is an indicator whether the code is operating efficiently. The thresholds, as listed in Table 1, are considered the limits where an investigation into the code section's vectorization would be useful. These limits are based on recommendations of Intel® [15] for the KNL architecture and while these hold true for most applications running on KNL, they are only guidelines and should be applied with care.

An additional metric, the `VPU intensity`, offers a rule of thumb on how well a loop is vectorized, calculating the proportion of vectorized operations on total arithmetic operations. This metric should be applied only to small pieces of code and certain non-arithmetic operations, such as mask manipulation instructions, are counted as vector operations, which can skew this ratio.

Table 1 defines the metrics as ratios of hardware counters provided by the KNL architecture. These can be accessed in Score-P through the PAPI metrics interface and can measured at a call-path level on each thread. To calculate all derived metrics, multiple native hardware counters have to be recorded. Since the KNL architecture provides only two general purpose counters per thread, multiple measurements have

Table 1 Vectorization metrics and their thresholds

Metric: L1 Compute to data access ratio	**Threshold:** < 1
`UOPS_RETIRED.PACKED_SIMD / MEM_UOPS_RETIRED.ALL_LOADS`	
Metric: L2 Compute to data access ratio	**Threshold:** $< 100*$ L1 Compute to data access ratio
`UOPS_RETIRED.PACKED_SIMD / MEM_UOPS_RETIRED.L1_MISS_LOADS`	
Metric: VPU intensity	**Threshold:** < 0.5
`UOPS_RETIRED.PACKED_SIMD /`	
`(UOPS_RETIRED.PACKED_SIMD + UOPS_RETIRED.SCALAR_SIMD)`	

to be used to obtain the full set of counters required. To facilitate a consistent user experience, the Score-P/Scalasca workflow has been extended to automate multiple runs with varying settings, which we describe in the following section.

5 Measurement Work-Flow

The analysis workflow for users of Scalasca comprises (1) instrumentation, (2) measurement, and (3) result examination. The analysis options presented in the previous sections require an adaption of this workflow, as multiple measurement runs need to be conducted before the results can be examined. Additionally, the results of the individual measurements need to be merged before examination to provide a holistic view across all of the different measurements. Furthermore, Cube needs to compute possible optimization candidates based on the unified results in an additional analysis step. To retain usability, we adapted the Scalasca toolset to automate the necessary measurements and post-processing steps.

5.1 Multi Run

The SCIPHI workflow needs multiple measurement runs, as the required hardware counters cannot be obtained by a single measurement. It is a known limitation of hardware performance counters that only specific combinations of counters can be combined in a single measurement [21]. The counters required for the analyses in SCIPHI need to be obtained in multiple measurements. So while the changes to the measurement work-flow are driven by the specific use case of SCIPHI, the generic implementation of the workflow adaptation also benefits other hardware counter measurements. Further benefiting scenarios are the quantification of variations in measurement, or the verification of statistical stability of results.

We implemented this adapted workflow as part of the Scalasca convenience command `scan`. As `scan` already automates the run of multiple execution steps—measurement and subsequent trace analysis—it is the natural target for also orchestrating multiple profile measurements. The basic mechanism of different execution settings per measurement is using environment variables. In principle, this allows arbitrary changes to the execution environment of each of the different runs, however, for the specific case of the KNL analysis the parameters have to be chosen carefully as the ability to merge the single run results depends on the similarity of the application runs. The new mode of multiple runs is controlled by input parameters of `scan`. The user has to specify the number of runs and the sets of variables for each run. Because the variables parameter can have a wide range of complexity, two ways of specification are offered: (1) a string combining all setting and (2) a configuration file.

Using a single string to specify the individual runtime parameters is convenient only for small parameter sets. Parameters within the string can be separated by two types of separators:`%` as run separator and `|` as key-value pair separator. An example for multiple key-value pairs is given below:

```
SCOREP_METRIC_PAPI=PAPI_TOT_CYC|SCOREP_TOTAL_MEMORY=33M
```

Such strings can be concatenated with an additional `%` between settings for each run, where the *i*-th sub-string indicates the environment settings for the *i*-th run. Empty sub-strings, indicated by `%%`, specify runs without special run configuration. If the configuration string contains less sub-strings as there are runs configured, `scan` uses as many of the sub-strings for its runs as available and assumes any missing sub-string at the end to be empty.

Using a configuration file is more practical when using a large number of variables or runs. A valid configuration file is a text file with one variable definition (key-value pair) per line and lines starting with a `%` as run separators. Analog to shell style comments lines starting with `#` as well as empty lines are ignored.

In either case per-run variables can only be used if they are not already set to a value in the current environment enabling the use of global and local variables and a strict separation of both. With these changes `scan` creates a single experiment directory with sub-directories containing the numbered results of the individual steps.

5.2 Cube Tools

Given successful measurements, further analysis is performed by Cube. In recent releases, the Cube GUI has been extended by a flexible plugin API [16], which allows the easy addition of new capabilities through feature-specific plugins. In the context of this work, Cube needed to be extended in two aspects. First, the management of multiple measurement results per run and their combination into a single unified result. Second, the analysis generating optimization suggestions based

on the criteria presented in the Sect. 4. Given their different nature, these aspects are supported through two separate plugins: *Spotter* and *KNL Advisor*.

The Spotter plugin manages the creation of a single Cube profile from the multiple measurements found in the measurement archive. It scans a given directory and merges any found Cube profile to the joint profile one after the other. Any new metric found in a single profile is added to the joint Cube profile. This way, all metrics existing in any of the profiles will be present in the joint Cube report containing all partial counter recordings. During the merge process metrics existing in all profiles like `time` are replaced by the last instance in the merge chain. Therefore, a measurement run without additional counters can be used to provide low overhead time information as suggested in Sect. 5.1. With the complete set of base metrics Cube can calculate the derived metrics described in Sect. 4.1. As counters from possibly different measurement steps are combined, the user must consider the results with care and place them in context of the application's deterministic and repeatable behavior.

The KNL Advisor processes the joint profile generated by the Spotter plugin and applies the thresholds defined in Sect. 4.1. Based on these thresholds it generates a list of possible candidates for optimization referring to code location and triggering metric for each incident. As mentioned before, automatically applying thresholds bears the risk of creating false or irrelevant information. As most of the metrics generally make sense only for small code regions with focus on loop optimization, the calculation of the metrics and their candidates is restricted to loops and their children in the call tree. This still allows the user to focus on the most relevant parts from a vectorization point of view and their respective sub-trees while reducing the clutter noticeably. Loops are special code regions marked during instrumentation and recorded during measurement. Score-P allows for measurement of two different loop constructs: (1) OpenMP loops, which as automatically detected during instrumentation, and (2) user-defined loops, which are manually instrumented using Score-P's user-instrumentation API. To narrow down the list of candidates even further, they are classified for relevancy. The metrics used as the basis for the analyses presented in this work are all relative values. These ratios may indicate a high relative impact while their absolute impact on the overall runtime may be negligible. Therefore, relative *and* absolute impact needs to be taken into account. To honor user knowledge about the code and allow for application dependencies no specific, absolute cut-off threshold has been chosen here. Instead, a percentage slider based on a code region's runtime in relation to the total runtime determines the list of current suggestions, which is updated when the percentage is changed. This allows the user to apply different levels of detail to inspect and choose possible candidates for optimization.

Figure 5 shows an example of the KNL advisor plugin in use. On the right side the panel lists the suggestions for the current percentage setting, chosen from the slider in the toolbar above, as a list of call site and triggering metric. This list contains suggestions concerning the three vectorization metrics presented in Sect. 4.1. The left and middle pane show relevant metric and call-path information, respectively, for the currently selected incident. Additionally, the call tree also highlights every other incident of the same metric in green. When selecting a different incident the call tree and the metric selection will update their selections to the relevant view.

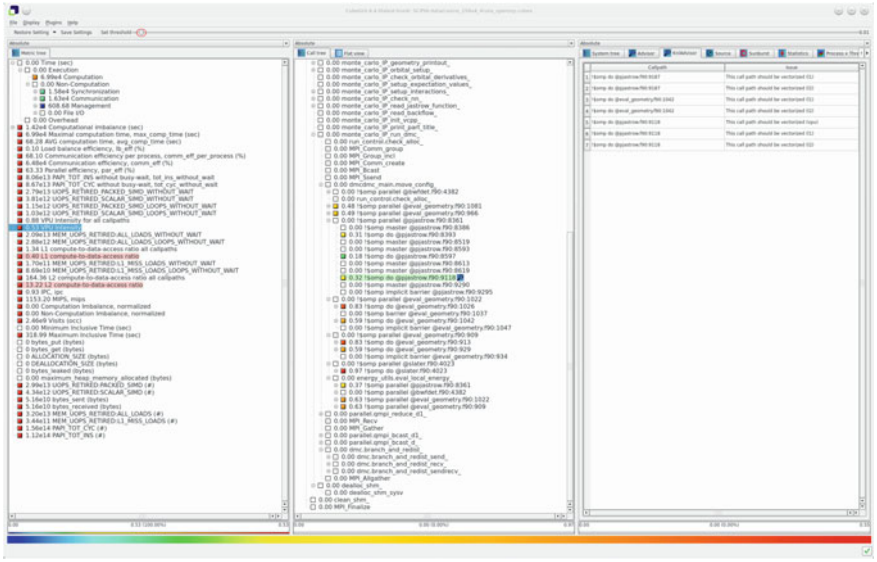


Fig. 5 KNL Advisor plugin

Generating optimization candidates and presenting them to the user in this way provides a intuitive starting point for the performance analysis. However, the user should always keep in mind that the suggestions given by the KNL advisor are only guidelines for optimization.

6 Topology Visualization

When examining the behavior of applications at large scale, it is important that the user understands how the individual processes and threads are distributed across the machine. A visual representation of the execution topology facilitates such understanding through an intuitive access to information about the execution context. Furthermore, in the case of many-core architectures, such as Intel KNL, the distribution on a single many-core node is of particular interest to gain the best performance for an application. The Score-P measurement infrastructure encodes this information as Cartesian topologies and saves it as part of the measurement profile to be visually explored using the Cube report browser.

Score-P Cartesian topologies map processes and threads to coordinates in multiple dimensional regular grids. While Score-P can record Cartesian topologies of arbitrary dimensions, they are folded onto three dimensions for visualization in Cube. Score-P records meta information about the dimensions of the underlying topology and coordinates for each location, however, it is currently limited to CPU threads and processes excluding accelerators.

Currently, Score-P supports the generation of Cartesian topologies from four sources: (1) MPI Cartesian communicators, (2) proprietary query interfaces, (3) processes-by-threads matrices, and (4) user-defined topologies. Through the interception of the `MPI_Cart_create` call, Score-P automatically uses the information passed to the call to generate all necessary topology information and no further user interaction is needed. Each call `MPI_Cart_create` will create a separate topology. For platforms that provide an interface to supply coordinate information, such as the IBM Blue Gene or the Futjitsu K Computer, Score-P creates a hardware topology, mapping processes and threads with relation to the given dimensional information. For hybrid applications, Score-P automatically generates a two-dimensional topology with all processes in one dimension and their respective threads in the other dimension. The relationships visualized are similar to those shown in the system tree widget. However, the data is presented in a much more concise fashion, allowing a better overview of larger configurations. Finally, Score-P provides a user API to manually define Cartesian topologies. This allows users to record topology information as needed and enables the generation of application-level topologies that do not directly map to any other method above.

As topologies are regarded static through-out the execution of the application, topologies require pinning of threads to cores. Since the coordinate mapping is only done once per location Score-P will produce erroneous results if a thread migrates during the run-time. To keep a consistent mapping between threads and hardware locations, over-subscription isn't supported and application threads should be evenly spread between hardware threads.

An application run without explicit use of topologies in either MPI or user instrumentation will show the "Process x Threads" topology and, if supported, a hardware topology. Users can enable and disabled each source of topology information through the use of environment variables. This can become useful in cases where hardware topologies are problematic or for example when MPI Cartesian topologies are created in a loop iteration creating too many for practical use in the Cube result. As the coordinate information is only acquired once per location the run-time overhead is limited, however these meta data will increase the size of the measurement results depending on the number of locations. Therefore, depending on scale and memory requirements it can be helpful to reduce the memory overhead caused by topologies.

As part of the project, we investigated options to provide platform topology support for the KNL architecture. Since, the KNL architecture doesn't supply a specific interface to inquire this information directly during measurement, a more generic approach has been implemented in Score-P. Using the third-party *hwloc* library [2], Score-P now provides topology information for generic Linux systems and the KNL architecture in particular. Hwloc gathers node-level information about the core distribution and the memory hierarchy. It is an Open-MPI sub-project, mostly developed by the TADaaM team at Inria. As it provides only node-level information and no further information about the network structure is available, Score-P maps all nodes onto a single dimension. A further limit of hwloc is that some levels of the collected hierarchy, e.g., the L2 cache level of the KNL tiles, have only a logical numbering, which precludes any implicit assumption about a direct mapping to the respective

hardware element. This limits the recording of direct neighborhood relations of the 2D tile map of the KNL chip, as shown in the schematic of Fig. 1. Therefore, Score-P maps it to a single dimension instead.

In general, topology data is stored in both Cube profiles as well as OTF2 trace definitions. However, in the context of this project, we focused on the user benefits of visualization in Cube, as the core of the extensions for SCIPHI are based on profile data.

Figure 6 shows an example of a hardware topology for KNL obtained via hwloc during a measurement of the NAS Parallel Benchmark [20]. The Cube topology viewer plugin is available in the third pane of the Cube display, as an alternative visualization option to system tree, boxplot and others. Every topology created during run-time is represented by an additional tab in this pane. With the help of the detachment mechanism for tabs, Cube allows the inspection of multiple topologies simultaneously. Most of the plugin space is used for a three dimensional display of the selected topology. The view can be manipulated in zoom and orientation either through mouse interaction or the toolbar. The user can select locations and query process and thread information on the selected coordinate. At the bottom of

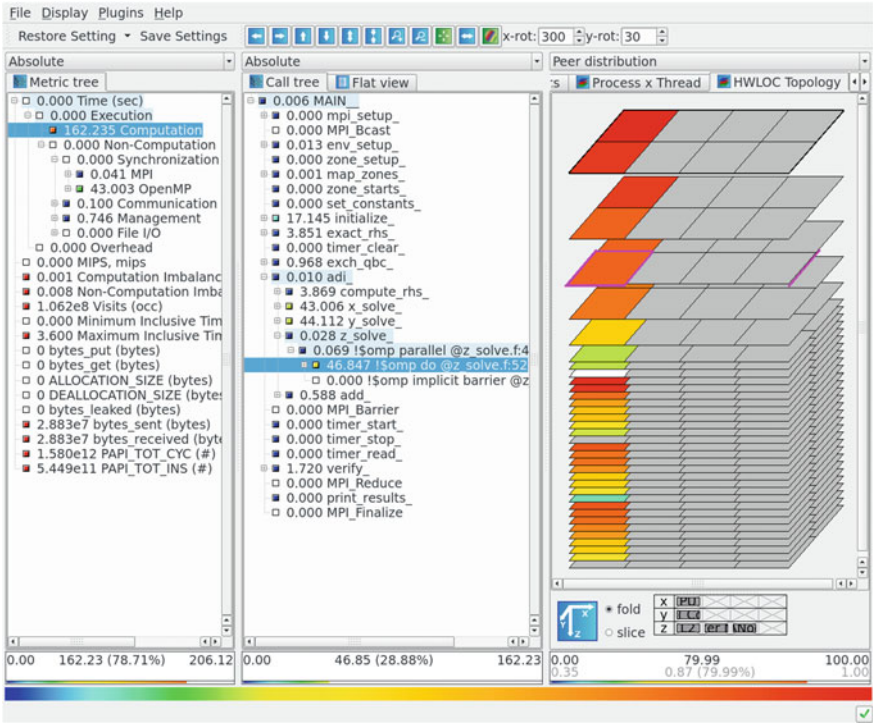


Fig. 6 Topology on a single node with one thread per core. Using one of four hardware threads per core and with two cores for each of the 36 tiles in z-dimension

the display is a control interface that allows dimensions to be mapped to the three dimensions of the display, which becomes a necessity once you have more than three dimensions defined in your topology. There are two options to reduce the number of dimensions: folding and slicing. With folding the user can choose which input dimensions should be folded into one output dimension. In the case of slicing, three dimensions are selected to be shown completely and for the remaining dimensions single elements are chosen. Figure 6 shows a single node example, where the cluster dimension doesn't provide additional information and can be safely merged with the tile level of the KNL architecture.

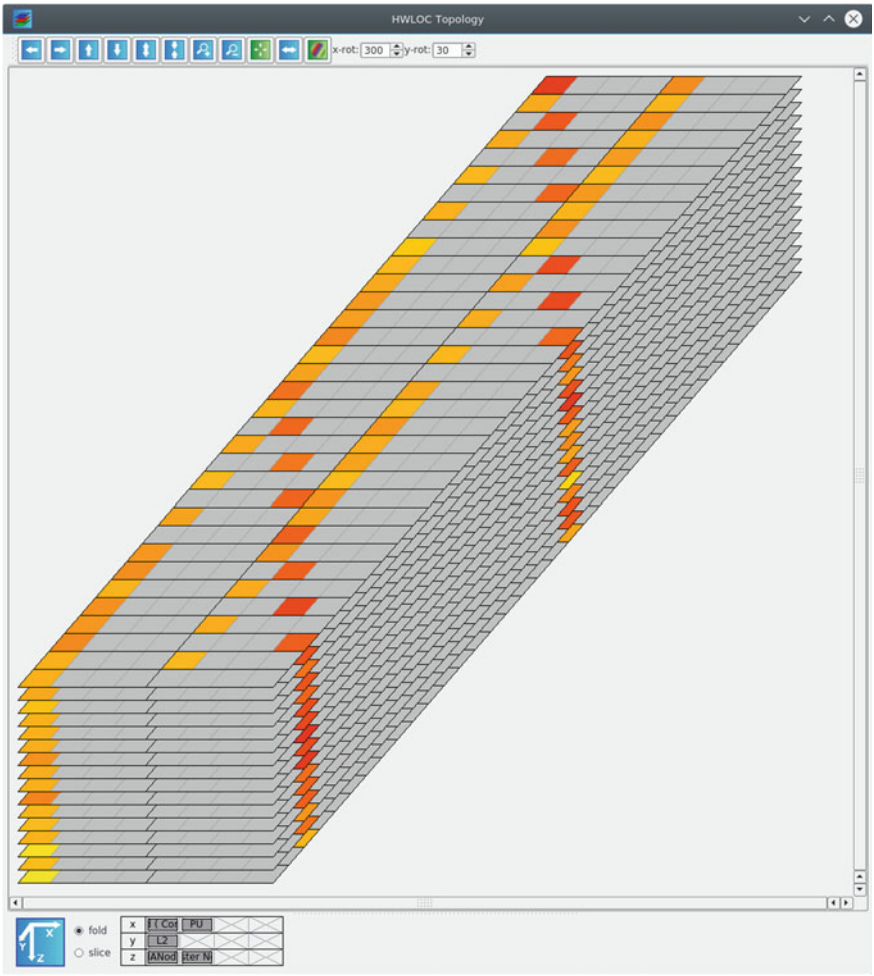


Fig. 7 View of the topology in detached state for an application run with a set of cluster nodes. On the x axis the two cores of a tile are arrayed in sequence with each of their four hardware threads grouped together. In combination with the 36 tiles in y direction each x, y plane represents a node. The 16 levels in z direction show the 16 used nodes for this run

That leads to an arrangement, where one (x,y) layer represents the two cores each with four hardware threads of a tile while the z-dimension shows the 36 tiles of a KNL node. Unused coordinates within topology are grayed out, as can be seen in this example where only one thread per core has been used.

To highlight the effect of different dimensions and their layout on the topology visualization, Fig. 7 shows the detached view of a second example of using topologies on KNL. This second example shows a topology representation of an application measurement run on multiple nodes. The application is a Monte Carlo simulation called Casino, executed on 16 nodes of the CINECA Marconi cluster [14]. As the measurement now spans multiple nodes, the off-node dimension cannot be folded into the node layer without obscuring the node-level information. Figure 7 therefore keeps this dimension separate in the z dimension, while retaining the node-level dimensions mapped to the x and y dimension. That way one x,y plane represents one KNL cluster node with a line along the x dimension containing one tile, showing the hardware threads grouped by core.

As these two simple examples show, there is not a singular best way to arrange the dimensions in three dimension. Its usefulness depends on run-time configurations and user focus. Specifically in the case of the KNL architecture, the number of relevant dimensions also depends on the chosen cluster and memory modes as they influence the number of NUMA and sub-NUMA nodes. Furthermore, the examples demonstrate that topologies are a powerful tool to visualize run-time distribution of performance metrics, across large-scale measurements while having all locations visible without needing to scroll through lists of locations. This enhances the user ability to recognize patterns based on process placement and severity.

7 Conclusion

In this work we presented a set of extensions for Score-P that originated from a cooperation with Intel® and were therefore focused on the KNL architecture. With objective of improving the user experience and providing options for more in depth intra-node analysis, we focused on the important topics of memory hierarchy and vectorization. For the memory hierarchy, the explicit use of the two types of available memory, DDR4 SDRAM and MCDRAM, were of particular interest. The extensions highlight possibilities for tracking allocations and actual usage as a guideline to steer the developer to an efficient use of the fast MCDRAM. The work on the vectorization assistance using knowledge about the specific target architecture presents a deviation from the standard Score-P metric definition. Instead of reporting just the severity of a metric, thresholds are provided as deciding factor to suggest optimization candidates.

The architecture dependent use case created possibilities for future work in broadening the focus of the presented extensions. The adaptations for multiple runs have a wide range of possible use cases, however for generic and integrated use in the standard work flow the limiting ramifications have to be addressed, in particular ensuring or at least checking the similarities between runs. Also, for a broader use of metrics

from different measurements the scaling of potential different timings have to be considered, possible through the use of reference counters. Streamlining the workflow might also contain an automatic way of source to source loop instrumentation to avoid the manual step of interaction.

Acknowledgements We would like to express our thanks to Intel Corporation, who supported this work by the Intel Gift Grant "SCIPHI—Score-P and Cube extensions for Intel PHI".

References

1. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: Hpctoolkit: tools for performance analysis of optimized parallel programs. Concurr. Comput.: Pract. Exper., **22**(6):685–701, April 2010 http://hpctoolkit.org
2. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a generic framework for managing hardware affinities in hpc applications. In IEEE, editor, PDP: The 18th Euromicro International Conference on Parallel, p. 2010. Distributed and Network-Based Computing, Pisa, Italy, February (2010)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3), 189–204 (2000)
4. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open trace format 2—the next generation of scalable trace formats and support libraries. In: Proceedings of the International Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30–September 2 2011, vol. 22 of Advances in Parallel Computing, pp. 481–490. IOS Press (2012)
5. Intel Corporation. Intel architecture instruction set extensions programming reference. https://software.intel.com/isa-extensions
6. Intel^{R} VTune^{TM} amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe
7. Jurenz, M., Brendel, R., Knüpfer, A., Müller, M., Nagel, W.E.: Memory allocation tracing with VampirTrace, pp. 839–846. Springer, Berlin, Heidelberg (2007)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set, pp. 139–155. Springer, Berlin, Heidelberg (2008)
9. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W,E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P—a joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proceedings of 5th Parallel Tools Workshop, 2011, Dresden, Germany, pp. 79–91. Springer, Berlin, Heidelberg, September 2012
10. Liu, X., Mellor-Crummey, J.: A data-centric profiler for parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 28:1–28:12. ACM, New York, NY, USA (2013)
11. Liu, X., Wu, B.: Scaanalyzer: a tool to identify memory scalability bottlenecks in parallel programs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, pages 47:1–47:12. ACM, New York, NY, USA (2015)
12. Lorenz, D., Böhme, D., Mohr, B., Strube, A., Szebenyi, Z.: Extending Scalasca's analysis features. In: Cheptsov, A., Brinkmann, S., Gracia, J., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2012, pp. 115–126. Springer, Berlin, Heidelberg (2013)
13. Mallinson, A.C., Beckingsale, D.A., Gaudin, W.P., Herdman, J.A., Levesque, J.M., Jarvis, S.A.: Cloverleaf: preparing hydrodynamics codes for exascale. In: A New Vintage of Computing: CUG2013. Cray User Group, Inc. (2013)

14. Marconi, new CINECA tier-0 system. http://www.hpc.cineca.it/hardware/marconi
15. Reinders, J., Jeffers, J., Sodani, A.: Intel Xeon Phi Processor High Performance Programming Knights, Landing edn. Morgan Kaufmann Publishers Inc., Boston, MA, USA (2016)
16. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4 From performance report explorer to performance analysis tool. Proced. Comput. Sci. **51**, 1343–1352 (2015)
17. Schöne, R., Tschüter, R., Ilsche, T., Hackenberg, D.: The VampirTrace Plugin Counter Interface: Introduction and Examples, pp. 501–511. Springer, Berlin, Heidelberg (2011)
18. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)
19. Treibig, J., Hager, G., Wellein, G.: Likwid: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of PSTI 2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA (2010)
20. Van der Wijngaart, R.F., Jin, H.: NAS Parallel Benchmarks, Multi-Zone versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, USA, July 2003. http://www.nas.nasa.gov/Software/NPB/
21. Wylie, B.J.N., Mohr, B., Wolf, F.: Holistic hardware counter performance analysis of parallel programs. In: Proceedings of the Conference on Parallel Computing (ParCo), Malaga, Spain, pp. 187–194, September 2005

Towards Elastic Resource Management

Isaías A. Comprés Ureña and Michael Gerndt

Abstract A new paradigm for HPC Resource Management, called Elastic Computing, is under development at the Invasive Computing Transregional Collaborative Research Center. An extension to MPI for programming elastic applications and a resource manager were implemented. The resource manager is an extension of the SLURM batch scheduler. Resource elasticity allows the resource manager to dictate changes in the resource allocations of running applications based on scheduler decisions. These resource allocation changes are decided by the scheduler based on performance feedback from the applications. The collection of performance feedback from running applications poses unique challenges for the runtime system. In this document, our current performance feedback system is presented.

1 Introduction

Distributed computing systems are expected to deliver performance that is commensurate to their available hardware resources. This is achieved by the optimization of system-wide performance metrics. The optimization of these performance metrics is a task usually delegated to schedulers. In the case of distributed systems, schedulers take as input the jobs to be performed and the set of available compute resources. They produce as output the job startup order and the resources where they will be executed. These orders are referred to as schedules. These schedules affect the per-

Support for this work was provided by the Transregional Collaborative Research Centre 89: Invasive Computing (InvasIC) [29].

I. A. Comprés Ureña (✉) · M. Gerndt
Technical University of Munich (TUM), Rheinstrasse 5,
80803 München, Germany
e-mail: isaias.compres@tum.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_7

formance of individual applications and whole systems, and therefore determine the quality of schedulers.

The terms resource manager and scheduler are sometimes used interchangeably. In reality, these are different software components that are often bundled together due to their equal importance. Distributed systems need both a resource manager and a scheduler in order to share their resources with multiple users in a fair and efficient manner.

This document begins with an introduction to the general multiprocessor scheduling, the batch scheduling and the runtime scheduling problems, to help illustrate the need for performance feedback on resource-elastic systems. The additional features of the resource manager that provide performance feedback to the scheduler are described afterwards.

2 Theoretical Background on Multiprocessor Scheduling

The general multiprocessor scheduling problem is stated in an abstract manner in this section. The problem statement for batch scheduling with static resource allocations is presented after that, together with a short discussion on the taxonomy of schedulers and how it is classified. This problem statement is then extended to fit the more specific elastic scheduling problem addressed in this work. New requirements are identified from the new problem statement.

2.1 Problem Statement

Multiprocessor scheduling is an optimization problem that can be stated verbally as follows: given a set of tasks to be completed and a set of resources that can complete them by some means, find an assignment of tasks to resources that optimizes a set of objective functions. The tasks are bounded in time and may require collectively more resources than are available simultaneously; therefore, the assignment of tasks to resources may also require an order. Different orders can produce different outputs of the objective functions.

We can define the problem of scheduling more rigorously. Let T be a set of tasks t_i where the subscript $i \in \mathbb{N}$ identifies each task uniquely; this set may or may not be finite. Similarly, let R be a set of m resources r_j where the subscript $\{j \in \mathbb{N} \mid j < m\}$ identifies each resource uniquely. One or more resources in R can perform the tasks in T in some manner. If $\tau(t_i) \in \mathbb{R}$ is the maximum execution time and $\rho(t_i) \in \mathbb{N}$ the number of resources required to perform a task t_i, then we can define multiprocessor scheduling as the following optimization problem:

$$\begin{aligned} &\text{given inputs} && T = \{t_i \mid \tau(t_i) < \infty \wedge \rho(t_i) \leq m\}, \\ &&& R = \{r_j \mid j < m\} \\ &\text{compute a} && S = \{t_i \mapsto \varrho_i\} \\ &\text{that optimizes} && \sum_{k=0}^{w} O_k \end{aligned} \tag{1}$$

The result of this optimization is a schedule S. The schedule is a set of mappings from individual tasks t_i into specific subsets of resources ϱ_i of size $\rho(t_i)$, where $\varrho_i \subset R$. Tasks where $\rho(t_i) > m$ are impossible to schedule and therefore not considered.

Objective functions typically produce single scalar values in $\mathbb{R}$ within the range $[0, \infty)$. By optimizing (either minimizing or maximizing) the sum of the output of each O_k objective function (e.g. idle node time), where $\{k \in \mathbb{N} \mid k < w\}$, the quality of the produced schedule can be improved. Different objective functions can evaluate the quality of full schedules S or individual mappings $t_i \mapsto \varrho_i$. This allows schedulers to optimize based on system-wide metrics, performance metrics of individual applications, or both.

The sum of all required resources of the tasks in T may exceed the total number of resources m in R:

$$\sum \rho(t_i) > m \tag{2}$$

In such a condition all tasks cannot be started simultaneously at the earliest time of the schedule $\{\delta_0 \in \mathbb{R} \mid \delta_0 > 0\}$. Because of this, both a starting time and duration need to be added as part of each mapping in the schedule when resource sharing is not allowed. Each mapping then becomes a reservation of resources with a starting time $\delta_i \geq \delta_0$ and the duration of its task $\tau(t_i)$, in addition to its set of unique resources ϱ_i. A schedule S then becomes:

$$S = \{t_i \mapsto \langle \varrho_i, \delta_i, \tau(t_i) \rangle\} \tag{3}$$

This modification to S can be inserted in the initial optimization problem definition (Eq. 1) to indicate that schedules need to be produced with these additional timing specifications.

2.2 Computational Complexity

The theoretical complexity of the multiprocessor scheduling problem can be determined with the aid of complexity theory. The goal is to determine the asymptotic complexity of the optimization problem based on its inputs. A bound to the number of steps of possible algorithms, based on the number of steps required to reach a solution, should be determined. Thankfully, this topic has been of great interest to researchers and results from previous analyses [10, 12, 17, 19, 21] are available.

The multiprocessor scheduling problem belongs to a family of problems that have no known solutions of polynomial or lower complexity [6, 7, 18, 31]. It is for this reason that current schedulers rely on approximation algorithms that are based on heuristics. These algorithms settle for solutions that are feasible but not necessarily optimal; the assumption is that in most cases adequate heuristics guide the approximations so that produced schedules approach optimal results, based on a set of objective functions.

2.3 *Resource-Static Scheduling in Distributed Memory HPC Systems*

A scheduling problem for specific compute systems, in a more concrete way, can be classified by several characteristics related to its set of tasks, its set of resources and its method used to generate the output schedule. There have been several efforts to create a taxonomy of scheduling problems [5, 13, 21, 22, 25]. The scheduling problem in distributed HPC systems is clearly defined [8, 9, 16, 23, 26] for current resource-static execution models. Current solutions consist generally of First-Come First-Serve (FCFS) batch scheduling with static allocations and backfilling.

Current supercomputing systems are usually shared among several researchers across multiple institutions. Individual tasks are submitted to these systems by its users, in the form of batch job definitions. The arrival rate of these job definitions can be modeled with the aid of queueing theory. Batch job definitions include their number of resources required, their priority and their maximum execution time, among several other aspects that may not be as important to schedulers. Batch job definitions are entered in a queue. This queue represents the input task set T of the optimization problem 1.

The resources of current supercomputing systems tend to be similar. In most systems, the hardware on each node is identical. There may be cases where the nodes have heterogeneity internally (e.g., in the form of accelerators). A node is abstracted as a single resource in most cases. This means that in spite of the growing amount of parallelism internally at each node, schedulers operate on full nodes, instead of subsets of cores or even accelerators where available.

The operation of schedulers is currently divided in two steps: batch scheduling and backfilling. The batch scheduling step scans a window of the job queue and attempts to start as many jobs as possible based on their priority. When a job cannot be started immediately, it may instead get a resource reservation in the future. Once this first step is done, the scheduler proceeds to the backfilling step: it attempts to start jobs that fit in the gaps of remaining idle resources. Jobs that are started during this second step should not delay the start of higher priority jobs that have reservations.

The general strategy is illustrated in Fig. 1. It presents a scenario with four nodes, a job queue of six jobs with a priority based order. In the illustration, a schedule is computed where job 4 receives a reservation later than jobs 5 and 6 due to the

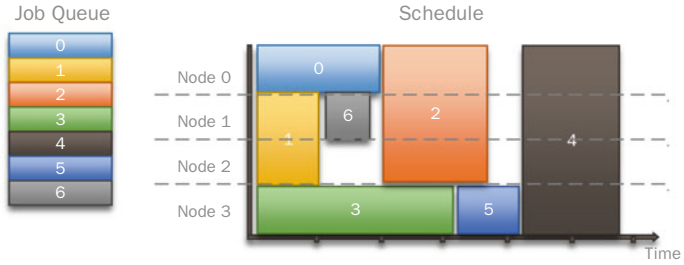


Fig. 1 Possible schedule of a set of static jobs ordered by priority in a queue

availability of resources. In the same schedule, job 6 is scheduled early to minimize idle nodes through a backfilling operation.

In summary, static batch scheduling with backfilling on current distributed systems has the following task set, resource set and algorithm properties:

- Task set:
  - Set properties:
    Multiple users submit tasks
    Tasks submitted randomly
    Unbounded task capacity
    Best effort First-In, First-Out (FIFO)
    Tasks are removed on completion
  - Task properties:
    Set of one or more tasks as jobs
    Jobs are time bounded
    Jobs and tasks are not periodic
    Fixed number of resources specified
    Jobs receive exclusive access to resources
    No Service Level Agreements (SLAs)
- Resource Set
  - Symmetric Multiprocessing (SMP) nodes as resources
  - Nodes have identical hardware (homogeneous)
  - Nodes may have attached accelerators
  - No quality of service (QoS) support
  - Resources are finite and cannot be scaled on demand
  - Resources are located in a single building
  - Power and energy scaling features available
  - No job or task migration support
  - No fault tolerance support

- Algorithm
  - Nodes as the units of resources
  - Job level scheduling (no task level scheduling)
  - Objective functions for mainly system-wide performance metrics
  - Two step *resource-static* scheduling
    Batch scheduling with priority based FIFO
    Backfilling to minimize idle nodes
  - Scheduling without performance guarantees
  - Scheduling without reactive adjustments
  - Jobs cannot be preempted

2.4 Modified Scheduling Problem for Resource-Elastic Execution

The scheduling problem described so far applies to cases where only static allocations are possible. Static allocations mean that the resource reservation of a job stays constant throughout its execution. The scheduling problem needs to be updated if the resource allocation of a job can change during the runtime of its tasks; resources may increase (expansion), decrease (reduction) or the unique nodes allocated to a job may change while their total stays the same (migration).

The current scheduling problem, solved with batch scheduling and backfilling, needs to be modified to include the added flexibility of resource-elastic execution. Only the properties of the jobs in the task set need to be modified:

- Jobs have a range of feasible resource counts.
- Jobs have a time bound that is a function of its resources.

This modified scheduling problem remains very similar to the preexisting one due to only these two differences. All other mentioned properties in the previous section remain. Jobs still retain exclusive access to the resources on its resource allocations, although some resources may be added or removed from this allocation at runtime. Due to this, the time required for the job to complete becomes dependent of the number of resources in time. In general, jobs will still provide a worst case time bound as part of its description.

Although similar to the preexisting scheduling problem, these two differences in the properties of jobs add new requirements to the algorithm of a potential scheduler. In addition to the previous batch and backfilling steps, a scheduler for HPC systems with resource-elastic execution capabilities must also:

1. Continuously monitor the performance of the tasks of running jobs.
2. Adjust the resource allocations of jobs based on their observed performance.

In the proposed design, the first activity is delegated to the resource management infrastructure, while the second activity is delegated to to a scheduler that is under

development. Most of the traditional batch-scheduling activities are still handled by a more traditional scheduler. In the remainder of this document, the way the first activity is carried out by the infrastructure will be described. The scheduler that is currently under development will not be covered in this document.

3 Performance Monitoring Infrastructure

The performance of individual jobs is monitored by the infrastructure. The infrastructure is composed of the MPI library and the resource manager components. Performance data is captured and a performance model is built. The performance model is then used to drive scheduling decisions.

The collection of data is performed in a hierarchical manner. At the lower level, each MPI library linked into each application process detects the structure of the computation in the local process and collects performance data. This structure is then reduced to a node-local representation by the `SLURMD` daemon at each node. Finally, the scheduler performs a final reduction to create the individual performance model of the distributed application. The set of models of all running applications are used to drive scheduling decisions.

3.1 Process-Local Pattern Detection and Performance Measurements

At the process-local view, the MPI library linked to the process performs pattern detection and performance metrics evaluations. The pattern of computation is detected before any performance metric is determined, since these metrics will be attached to specific control flow locations only after they are detected. Process local operations are kept to a minimum once a pattern is detected.

3.1.1 Pattern Detection

Since the pattern detection is intended to occur during the actual production run of applications, the minimization of its performance impact is of great importance. Because of this, the structure of computation is detected based on markers introduced by the compilation wrappers provided by the MPI library (`mpicc` and `mpifc` in this case). There have been previous works that rely on backtracing the sequence of calls in a program to determine unique locations during execution. These are then used as identifiers for pattern detection [1–4, 11, 15], such as loops, in MPI applications. The introduction of these markers at compilation time eliminates the overhead related to backtracing, although the technique is limited to binaries generated within a single software project.

The markers are inserted by splitting the compilation of objects into the emission of assembler and the final assembly step. Thankfully, most modern compilers have support for these operations. In the current implementation, the compiler wrapper works with Intel and GNU compilers. Versions 10.0 and later of the Intel compilers were tested, while versions 4.9 and later were tested for the GNU compilers. Other compilers were not tested, since those are the ones available in the SuperMUC system where this work was evaluated.

The current wrapper based technique relies on the way these compilers generate library calls in the emitted assembler. The actual API names of library calls are preserved, when linking C based libraries. Fortunately, MPI is a pure C based library and its calls can be easily identified with text processing in the intermediate assembler of each target object of the compilation. Additionally, since the MPI standard requires that any operation with the `MPI_` prefix be provided only by the MPI library in compliant programs, it is guaranteed that only MPI operations will be intercepted. Additionally, the `PMPI_` pattern can be selected to preserve support for any PMPI based profilers and tools.

Once the MPI calls are identified in the assembler code, a unique ID is computed and inserted before the MPI call through an operation available in the MPI library. This operation is currently called `MPIX_T_set_call_site_identifier`, and as its prefix `MPIX_T` suggests, it is a non-standard addition to the MPI tools interface. This tooling call sets the identifier for the device layer of the layered software architecture inherited from MPICH. This operation sets an integer identifier that is later read by the library at each individual MPI operation. This identifier establishes the uniqueness of the call site without the need of backtracing.

The MPI library relies on these markers to detect the structure of the computation at runtime. There have been several algorithms developed to detect patterns in sequences [14, 20, 24, 28, 30]. A pattern detection algorithm, that was originally designed to analyze programs from decompilation, fits well this pattern detection use case [32]; this algorithm is also used in other recent related works [2].

The pattern detection algorithm was implemented within the MPI library. In the current implementation, the algorithm provides the following output information to the runtime system, based on the current partial sequence of call site identifiers provided to it as input:

1. The detected Control Flow Graph (CFG).
2. Each node of the CFG is annotated with its number of revisits.
3. Nodes that are the heads of unique loops are marked.
4. Nodes that are the tails of unique loops are marked.
5. Nodes that are reentry points from nested loops are marked.

The CFG update routine is called at relevant MPI operations with their unique identifiers and types. There are different operation types for point-to-point, one-sided, collectives, MPI-IO, etc. The MPI library has an operation that serializes its local CFG to a shared memory segment, where it can then be read directly by the local daemon. Unique blocks of shared memory are dedicated to each MPI rank in the node.

An example can be used to better explain the algorithm's behavior. Listing 1 shows the log output of a single MPI process given the sequence of identifiers:

```
2 0 6 3 1 6 3 1 6 3 1 9 7 9 7 3 1 6 3 1 6 3 1 9 7 9 7
```

The detector can produce a text representation of its current CFG, in tabular form, as logging output. Listing 2 shows the detected CFG that matches the previous sequence. Each output row represents a node in the CFG. The first column is the address in the local memory of the process. The second columns is the identifier number. After that, the loop head flag (H), the loop body flag (B), the reentry counter (Re) and the revisit counter (Rv) are provided. The final two columns provide the tail data of loop heads, and the head data of loop body nodes. As seen in Listing 1, there is also a time differential (TD) computed at each step. In the current implementation, the time resolution of this differential is in nanoseconds. The time of creation is set each time a new node is added to the CFG. Total differential times from head nodes are accumulated on node revisits. The average distance in time from the head node of a loop to any node in the body can therefore be computed by dividing the accumulated differential by its total number of revisits.

```
 0: root id: 2
 1: id: 0; detected: 0; -> NOT in a loop; (TD: 4638)
 2: id: 6; detected: 0; -> NOT in a loop; (TD: 10243)
 3: id: 3; detected: 0; -> NOT in a loop; (TD: 14440)
 4: id: 1; detected: 0; -> NOT in a loop; (TD: 17938)
 5: id: 6; detected: 1; -> head: 6; (TD: 22178)
 6: id: 3; detected: 1; -> head: 6; (TD: 26174)
 7: id: 1; detected: 1; -> head: 6; (TD: 30090)
 8: id: 6; detected: 1; -> head: 6; (TD: 33756)
 9: id: 3; detected: 1; -> head: 6; (TD: 37407)
10: id: 1; detected: 1; -> head: 6; (TD: 41180)
11: id: 9; detected: 0; -> NOT in a loop; (TD: 44758)
12: id: 7; detected: 0; -> NOT in a loop; (TD: 48493)
13: id: 9; detected: 1; -> head: 9; (TD: 52336)
14: id: 7; detected: 1; -> head: 9; (TD: 56155)
15: id: 3; detected: 1; -> body re-entry; head: 6; (TD: 60054)
16: id: 1; detected: 1; -> head: 6; (TD: 63853)
17: id: 6; detected: 1; -> head: 6; (TD: 67418)
18: id: 3; detected: 1; -> head: 6; (TD: 70916)
19: id: 1; detected: 1; -> head: 6; (TD: 74361)
20: id: 6; detected: 1; -> head: 6; (TD: 77798)
21: id: 3; detected: 1; -> head: 6; (TD: 81239)
22: id: 1; detected: 1; -> head: 6; (TD: 84788)
23: id: 9; detected: 1; -> head re-entry; head: 9; (TD: 88710)
24: id: 7; detected: 1; -> head: 9; (TD: 92452)
25: id: 9; detected: 1; -> head: 9; (TD: 96131)
26: id: 7; detected: 1; -> head: 9; (TD: 99669)
```

Listing 7.1 Step by step updates based on the specified ID sequence

Figure 2 presents a graphical depiction of the text based CFG output. Reverse arrows on the left side of the figure represent loops, while the reverse arrow on the right represents a reentry. The time taken at each MPI block is represented as its vertical length. The time of the compute blocks can be computed by subtracting the MPI times from the differential from preceding MPI operations. Their time is also represented by their vertical length in the figure. In summary, all necessary data is included so that such a graph can be computed by the local daemon from the serialized CFG data.

```
Current detected Control Flow Graph (CFG):
0x03; id: 2; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x2b; id: 0; H: 0; B: 0; Re: 0; Rv: 0; tail:      ; head:
0x31; id: 6; H: 1; B: 0; Re: 0; Rv: 4; tail: 0x3d; head:
0x37; id: 3; H: 0; B: 1; Re: 1; Rv: 5; tail: 0x55; head: 0x31
0x3d; id: 1; H: 0; B: 1; Re: 0; Rv: 5; tail:      ; head: 0x31
0x4f; id: 9; H: 1; B: 0; Re: 0; Rv: 3; tail: 0x55; head:
0x55; id: 7; H: 0; B: 1; Re: 0; Rv: 3; tail:      ; head: 0x4f
```

Listing 7.2 Example CFG detected based on the specified ID sequence

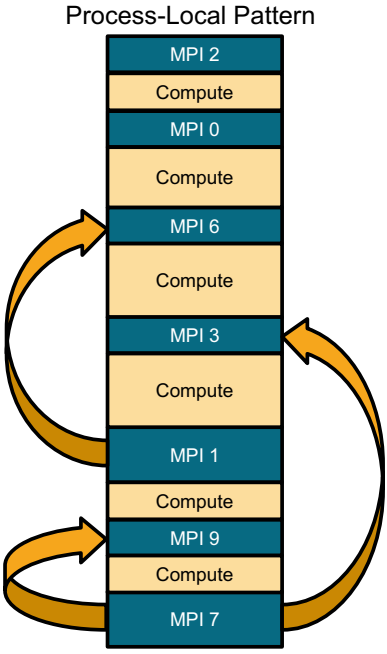


Fig. 2 Process-local Control Flow Graph (CFG) representation

3.1.2 Performance Measurements

The MPI library starts to record performance data once the heads and tails of one or more loops are detected. Currently two performance metrics are recorded:

1. Total Loop Time (TLT)
2. Total MPI Time (TMT)

The TLT metric is the total time spent on the detected loop. The TLT metric can be computed at each loop, including nested loops. The TLT metric is computed from two real numbers. The first one is its creation time. This time is set for each node in the CFG structure regardless of its type. The second one is the last visit time. The MPI library does not perform any more operations for this metric. Instead, the data is provided as it is to the local daemon once requested. The daemon is expected to perform the subtraction of these values for the total accumulated time, and to divide this value by the number of visits (revisits plus one) to get the average.

The second metric is the Total MPI Time (TMT). The TLT is inclusive of this time. This time is the difference between the entry and the exit times of each MPI call. In contrast to the TLT, these times are not stored in the CFG nodes where they are computed; instead, this metric is aggregated in the loop head of the node. There is no recursive search for the loop head in nested loops. The average can be computed by dividing the aggregated times by the total number of visits to their loop heads.

3.2 Node-Local Reductions and Performance Data Updates

Once a loop is detected, the library switches to a mode of CFG verification and performance data collection. As mentioned before, each process serializes its CFG data on its own shared memory segment. Each process notifies its local daemon on the following events:

- Loop detected
- Unexpected Loop exit
- Unexpected loop reentry

These events occur in the sequence presented in Listing 1: a loop detection occurs in steps 5 and 13, in step 11 an unexpected loop exit occurs, and in step 15 an unexpected loop reentry is encountered. All of these create changes in the CFG and therefore need to be communicated to the local daemon. These events tend to be more common during the initialization of MPI applications, and settle after a while. Expected loop reentries in the body or loop heads do not generate any events, since they do not trigger changes in the CFG. The library instead continues updating performance data without notifying its local daemon, if there are no changes to the CFG.

The number of notifications to the local daemon is limited by the sampling timer that currently defaults to one minute. This minimizes synchronization overheads, especially during the initialization of an application. If one or more loop detection or break events occur between timers, the local daemon is notified only once.

Performance data is updated separately from the CFG. These are updated periodically on each expiration of the sampling timer. These are only produced at the next loop head reentry, and not in any arbitrary MPI operation. Each metric specifies the identifier of its loop head, since more than one loop may be detected.

The local daemons do not read the performance data periodically. Instead, the latest data is read on demand when requests from the scheduler are received. These requests also have a field that optionally specifies a new value for the sampling timer. This enables the scheduler to adjust the frequency of data collection per application, based on previous performance data and trends.

3.2.1 Node-Local CFG Reduction

The daemon of a node keeps track of the notifications generated by each of its MPI processes. When any of its local processes have notified that their CFGs have been updated, it proceeds to read them and to perform a CFG reduction operation. The reduction operation depends on the order and type of the operations in it.

The following rules are followed on the collection of CFGs to produce a reduction:

1. Nodes outside of loops are ignored.
2. Consecutive point-to-point or one-sided operations are collapsed.
3. Identical loops are combined into one with a process range.

The reduced CFG is then stored in the memory of the local daemon. It is populated with performance data before it is sent to the scheduler on each request. If a request is received from the scheduler, but the CFG data is still unavailable, the response to the request has a field to indicates this.

An example set of four CFGs is presented in Fig. 3. All processes contain the loop from 6 to 1, but miss the nested loop with head 9 and tail 7. Rule 1 ignores the nodes 2 and 0. MPI operations with identifier 6 and 3 are of the type point-to-point. This means that they will be collapsed according to rule 2. All other operations are in loops. Finally, given rule 3, the loop from 6 to 1 will be clustered for ranks 0 through 3, while the loop from 9 to 7 will be separated for only rank 0. The information on its reentry is preserved. This indicates that it is nested within the common loop, but only at rank 0. The result is presented in Fig. 4.

The three rules in the reduction algorithm can be justified. The first rule is justified by the fact that code that occurs outside of loops is not relevant to elastic execution. The second rule comes from the observation that MPI applications that use multiple point-to-point and one-sided operations match logically across ranks. For example, it is common to observe branching based on the rank number of the local process in an MPI program to determine is the process will perform a send or a receive. These

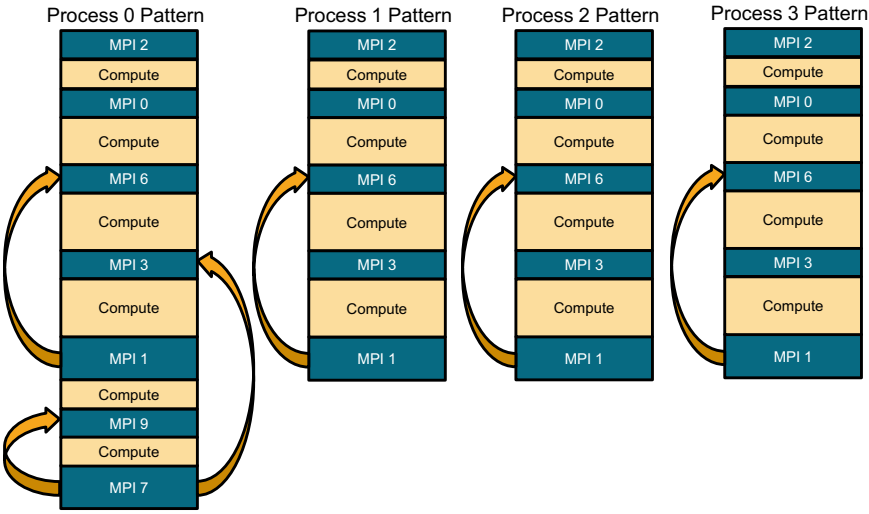


Fig. 3 Set of four CFGs at a node before reduction

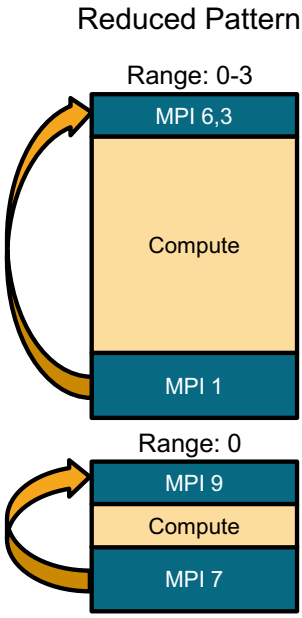


Fig. 4 Reduced CFG from Fig. 3

sends and receives can be matched as a single block of communication in a distributed view of the program, greatly simplifying the loop matching algorithm. This approach does not cover all possible patterns of point-to-point communication, and needs to be updated as the prototype matures. The final rule produces the reduction based on similarity. It is essentially a form of compression.

3.2.2 Node-Local Performance Data Reductions

The sum of all the TLT and TMT metrics of each process in a loop are added to the data of the reduced loop head nodes. In contrast, the mode (the value that occurs the most) of the loop revisit counts are set. It is expected that with enough revisits a small difference in the number of measurements will not affect the mean of the metrics significantly.

3.3 Distributed Reductions and Performance Models

The scheduler generates requests for performance data that reach all the daemons of an application. The requests and responses are routed through the `SRUN` binary of the application, over the Tree Based Overlay Network (TBON) that it creates with the nodes of its application. In the response to these request, each daemon sends the reduced CFGs with the TLT and TMT metrics attached to each loop head. The final distributed view of the CFG of the application is then generated from these at the scheduler.

Matching loops are reduced by combining all of their TLT and TMT metrics. The union of the process sets is set as the final range. The final distributed representation of the earlier example is presented in Fig. 5.

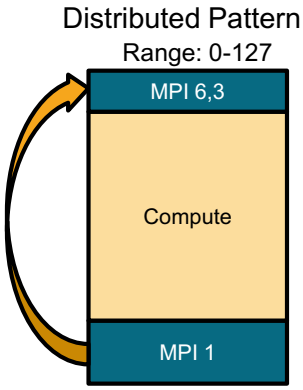


Fig. 5 Final reduced CFG at the scheduler from Fig. 4

Finally, the average loop time and MPI time metrics are computed based on the number of iterations of the loop heads and the TLT and TMT metrics provided. Additional memory is dedicated to store the mean, variance, minimum and maximum values of these final metrics. Finally, a vector of their recent values is stored, to detect performance trends.

3.3.1 SPMD-Phase Performance Model

Currently only one type of performance model has been implemented: the SPMD-Phase model. When the system detects one or more distributed loops, it creates an SPMD-Phase performance model instance for the application. Applications that do not fit this model (i.e., that have no distributed loop) are currently ignored. SPMD-Phase models consist of a set of distributed loops and their performance metadata. In general, models are used by the scheduling heuristic to try to ensure that application phases remain within their efficient range of resources.

4 Elastic Performance Feedback Overheads

A selection of resource manager operations is evaluated in this section. This selection contains all operations that impact the performance of MPI operations during normal computations. The operations that were not included are very numerous, but are either performed locally by one of the resource manager components, or do not impact the performance of preexisting MPI processes thanks to the latency hiding features of the design.

The evaluation has been performed in the SuperMUC [27] petascale system. This supercomputer is managed by the Leibniz Supercomputing Center (LRZ) and is located in Garching, Germany. The resources of this HPC system are managed by an IBM Load Leveler resource manager.

There were some challenges encountered when testing the custom resource manager and communication library. As may be expected, it is not possible to replace the preexisting resource manager. The new resource manager and MPI library were setup dynamically within a job. In that sense, the SLURM resource manager was nested inside of a Load Leveler job.

The SuperMUC system has multiple types of nodes divided in two sets: Phase 1 with Sandy Bridge CPUs, and Phase 2 with Haswell CPUs. Phase 1 nodes are based on a dual socket board with two Sandy Bridge-EP, Xeon E5-2680 CPUs. Each of these CPUs has 8 physical cores each, for a total of 16 per node, running at 2.7 GHz. Phase 2 nodes are also based on a dual socket board but with two Haswell-EP, Xeon E5-2697 CPUs. These have a higher CPU count of 14 physical cores each, for a total of 28 per node, running at lower 2.6 GHz.

All components (SLURM, MPICH and test applications) have been compiled with the GCC version 6 module provided in the SuperMUC system. The SuperMUC interconnect is based on Mellanox Infiniband network interfaces.

4.1 Tree Based Overlay Network (TBON) Latency

Resource adaptation instructions are set by the scheduler for each application when necessary. These adaptation instructions are probed by MPI applications periodically at locations where they can perform a redistribution of their domain. The communication between `SRUN` and the `SLURMD` daemons that manage the execution of an MPI application is important for the probe operation when the adaptation flag is set to true. The algorithm for probing has two sides: the side at each MPI process and the side at each `SLURMD` daemon. When the adaptation flag is set to true, multiple synchronization operations between the `SRUN` program and each daemon take place. These synchronization operations are performed over the Tree Based Overlay Network (TBON) that connects `SRUN` to each `SLURMD` daemon. Because of this, the latency of messages over the TBON can impact the overhead of MPI processes when they are required to adapt.

Figure 6 presents the latency of a single message and its confirmation from each participating node. In the figure, its scalability based on process count is presented. This means that the results for the Sandy Bridge and Haswell nodes will differ mainly due to the different core counts in the nodes. In the case of Haswell, only 20 nodes are needed to run 512 processes, while 32 nodes are needed in the Sandy Bridge nodes. As expected of a TBON network, the latency of messages scales logarithmically.

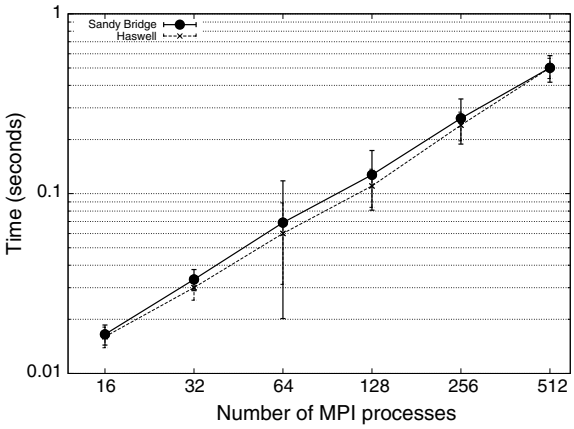


Fig. 6 Latency of TBON messages from `SRUN` to daemons

4.2 Control Flow Graph (CFG) Detection Overhead

In this section, the overhead of the set of operations that perform Control Flow Graph (CFG) detection is measured. Some of these operations impact the performance of MPI processes directly, while some can have a small impact since they are performed in the core where the `SLURMD` daemon of the node runs. These operations are: insertion, reduction, packing, unpacking and collapse.

The reduction, packing, unpacking and collapse operations are not as significant to the performance of MPI application processes due to their infrequent executions, as mentioned. That leaves the insertion operation as the only one that can impact the performance of application processes. The measurements are presented based on their scalability with respect to the size of the CFG graph, the total number of processes at each node, and finally the number of iterations of the loop in the application.

4.2.1 Scaling with Control Flow Graph (CFG) Size

It is important to understand how the detection overheads scale with increased CFG complexity. Figure 7 presents the scalability of all of the operations for CFG sizes between 8 and 1024 entries. Results for Phase 1 and Phase 2 nodes are included side by side for comparison. The sizes of CFGs are typically less than 100 entries, so the wide range of up to 1024 entries is pessimistic.

As mentioned before, the insertion latency is the most significant overhead. Unfortunately, the insertion latency scales exponentially with the number of entries in the CFG. Fortunately, although with bad scalability, the actual cost of the operation is small. A typical MPI operation runs for multiple milliseconds, while the insertion overhead is of around 700 nanoseconds for a 8 entry CFG, up to 10 μs for the extreme case of 1024 CFG entries. For the typical case of 128 CFG entries, the overhead of insertion is less than 2 μs.

The CFG reduction operation scales exponentially with the number of entries in the CFG. The overhead of 5 μs for 8 entries up to about 500 μs in the extreme 1024 entry case are acceptable, given the infrequency of this operation. The packing, unpacking and collapse operations scale exponentially, but their actual costs is much lower than the reduction operation, since these are performed in parallel with the participation of each MPI process. Their maximum cost of 100 μs at the extreme case of 1024 entries is also acceptable given the infrequency of these operations.

4.2.2 Scaling with Process Counts

In addition to scaling with the size of the CFG, it is also important to evaluate how the overheads scale with increasing numbers of processes at each node. These are intra-node operations, so only process counts that are expected to be possible, without oversubscription, in near future HPC nodes are considered: from 2 to 128 processes.

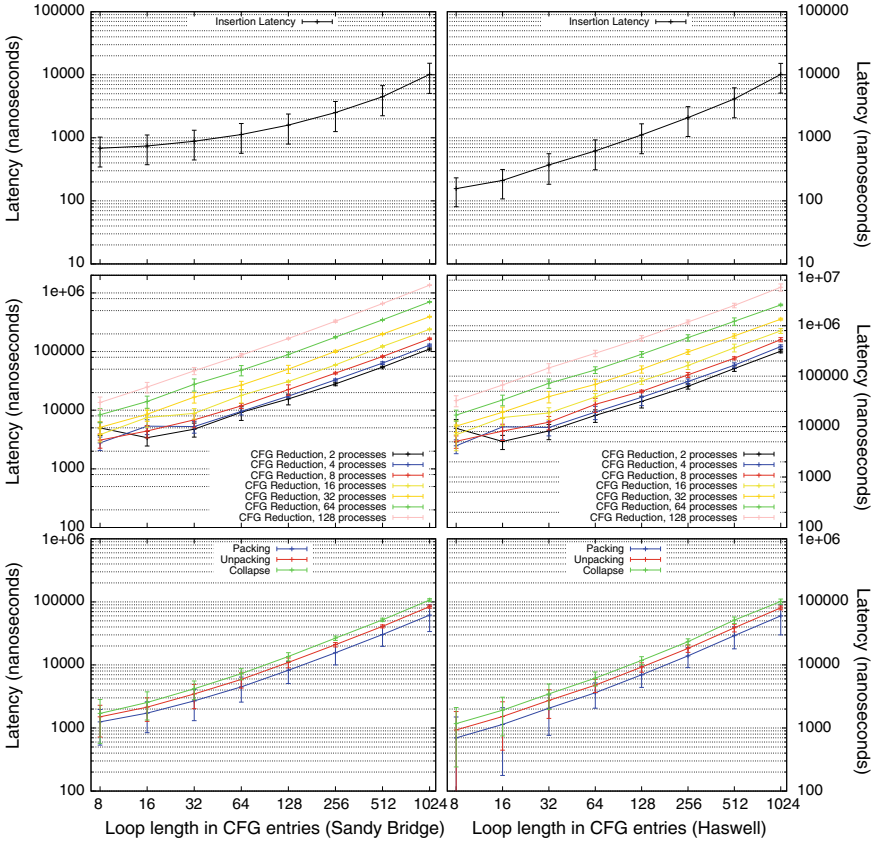


Fig. 7 CFG size performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

Figure 8 presents scalability data for the detection operations based on process counts. Results for the larger CFG sizes 256, 512 and 1024 are presented for Phase 1 (left) and Phase 2 (right) nodes. As can be seen, the overheads for the insertion, packing, unpacking and collapse operations do not depend on the process counts, while the reduction operation does. Their latencies vary between a few hundred nanoseconds to a few hundred microseconds.

Not scaling with the number of processes is desirable, since it means that an arbitrary number of processes can be added at each node and these overheads will not increase. This is specially important in the case of the insertion latency, since this overhead is added to each MPI operation while the CFG detection mechanism is enabled. Once the CFG logic switches to verification, this overhead is removed. The packing, unpacking and collapse overheads are not as impactful to application performance, as mentioned before, since these occur infrequently.

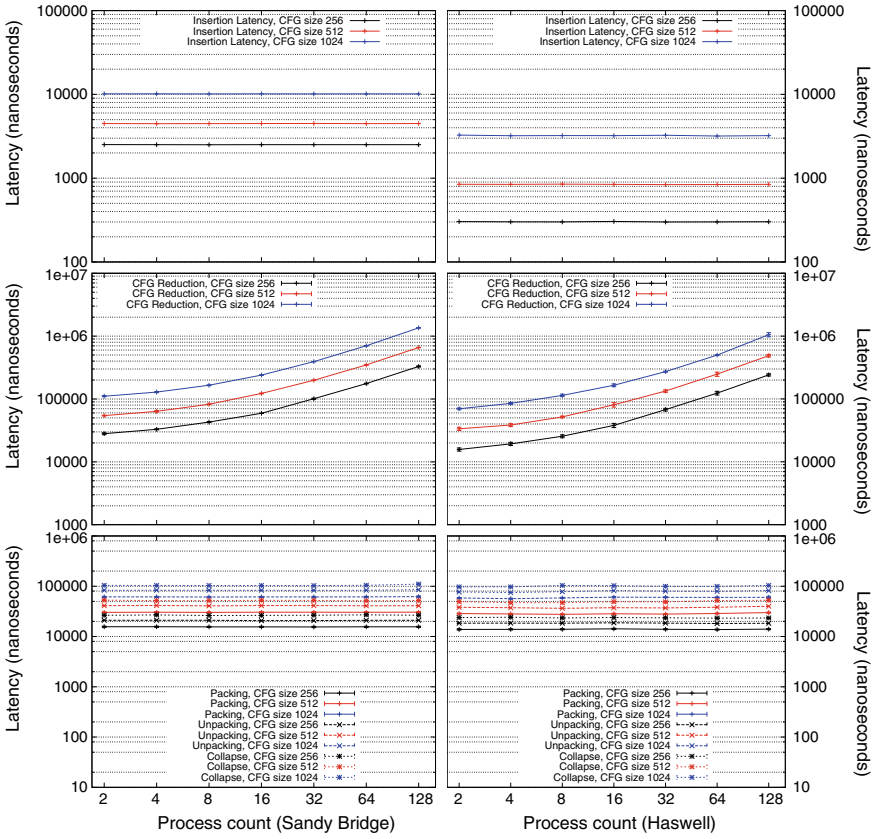


Fig. 8 Process count performance scaling. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

The situation for the reduction operation is not so fortunate, where its overhead increases with the number of processes per node of an application. As measured before, the overhead of this operation also increases with larger CFG sizes. Because of this, this operation has the worst scaling properties of the measurement infrastructure. Fortunately, these operations do not occur frequently and the absolute latency numbers it reaches are still not large.

4.3 MPI Performance Impact of the CFG Detection Overhead

Additional measurements were performed to evaluate the impact of these operations in actual MPI operations. MPI operations can run from a few microseconds to multiple seconds, depending on the type of operation, the number of processes and the size of the buffers.

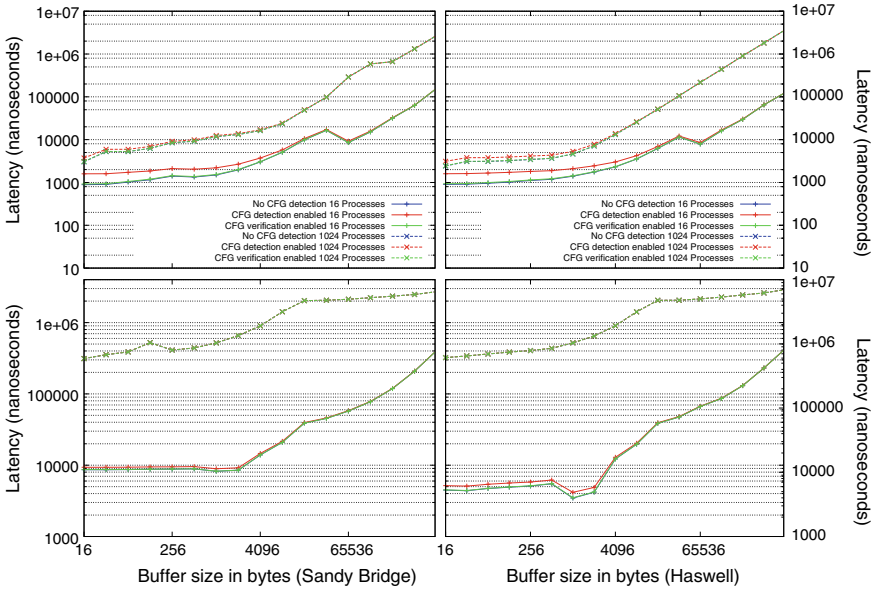


Fig. 9 `MPI_SEND` (top) and `MPI_BCAST` (bottom) performance examples with detection enabled and disabled on a 32 entry CFG loop. Results for SuperMUC Phase 1 (Sandy Bridge, left) and Phase 2 (Haswell, right) are presented

In Fig. 9, results for the `MPI_SEND` and `MPI_BCAST` operations are presented. These two operations were selected since they have the lowest latencies among the set of point-to-point and collective operations, respectively. The figure presents the latency for the `MPI_SEND` operation at the top and the `MPI_BCAST` operation at the bottom. Results for Phase 1 (left) and Phase 2 (right) nodes are presented side by side for comparison. Results for 16 and 1024 processes are presented with buffer sizes from 16 bytes up to a megabyte. The size of the CFG was set to 32 for these tests. Most applications and benchmarks that have been evaluated generate less CFG entries by the time they terminate.

As can be seen in the plots, the performance of `MPI_SEND` is only impacted significantly for message sizes of up to 4096 bytes, but only at lower process counts. For the case of 1024 processes, the overhead of the CFG detection algorithm is insignificant even for very small messages of 16 bytes. Additionally, the overhead of detection is not measurable on verification mode. This means that its overhead will only be observed when the detection algorithm has not encountered a loop, or when it exits a loop and resumes its detection.

A smaller performance impact can be observed for the `MPI_BCAST` operation. As mentioned before, the latency of this operation is the lowest among MPI collectives; therefore, the impact of CFG detection can be expected to be almost negligible when collectives are being used. Although the detection overhead is lower in terms of absolute latency, the percentage impact is higher in the case of Phase 2 nodes.

5 Conclusion

A CFG detection algorithm was implemented without the need of backtracing, in the MPI library. These CFGs are detected at each process and shared with the local resource manager daemons at compute nodes. These are eventually transferred to the scheduler running at a remote node through the TBON of the nodes allocated to each application. The overhead was shown to depend on the length of the CFG of applications. Because most applications produce CFGs that are in the order of hundreds of elements and the detection does not rely on backtracing, the overhead of detection was kept in the order of nanoseconds in most cases. The library switches to a verification only mechanism when a partial CFG remains stable. The overhead of verification cannot be measured even on single byte MPI messages with latencies in the order of microseconds.

A performance model can be produced with the data to drive scheduling decisions. It is expected that the integration of programming models and resource managers will increase in importance as exascale levels of performance are reached in HPC systems. Programming models that support resource-elastic execution and bring computational and energy efficiency benefits, while at the same time allowing for fault-tolerance, are expected to increase in importance in the near future. Performance feedback mechanisms, such as the one presented here, will allow future schedulers to make quality resource-scaling decisions to further improve system-wide efficiency metrics in HPC systems.

References

1. Aguilar, X., Fürlinger, K., Laure, E.: MPI trace compression using event flow graphs. In: Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25–29, 2014. Proceedings, pp. 1–12. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-09873-91
2. Aguilar, X., Fürlinger, K., Laure, E.: Automatic on-line detection of MPI application structure with event flow graphs. In: Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, pp. 70–81. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-06
3. Aguilar, X., Fürlinger, K., Laure, E.: Visual MPI performance analysis using event flow graphs. Proced. Comput. Sci. **51**, 1353 – 1362 (2015). https://doi.org/10.1016/j.procs.2015.05.322. URL http://www.sciencedirect.com/science/article/pii/S1877050915011308
4. Aguilar, X., Fürlinger, K., Laure, E.: Event flow graphs for MPI performance monitoring and analysis. In: Tools for High Performance Computing 2015: Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany, pp. 103–115. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-39589-08
5. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng. **14**(2), 141–154 (1988). https://doi.org/10.1109/32.4634
6. Coffman, E.G., J., Garey, M.R., Johnson, D.S.: An application of bin-packing to multiprocessor scheduling. SIAM J. Comput. **7**(1), 1–17 (1978). https://doi.org/10.1137/0207001

7. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011). https://doi.org/10.1145/1978802.1978814
8. Etsion, Y., Tsafrir, D.: A short survey of commercial cluster batch schedulers. Sch. Comput. Sci. Eng. Hebr. Univ. Jerus. **44221**, 2005–13 (2005)
9. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling—a status report. In: Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing, JSSPP 2004, pp. 1–16. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/114075221
10. Fortnow, L.: The status of the P versus NP problem. Commun. ACM **52**(9), 78–86 (2009). https://doi.org/10.1145/1562164.1562186
11. Fürlinger, K., Skinner, D.: Capturing and visualizing event flow graphs of MPI applications. In: Euro-Par 2009—Parallel Processing Workshops: HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Delft, The Netherlands, August 25–28, 2009, Revised Selected Papers, pp. 218–227. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14122-526
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)
13. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. In: Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications, Annals of Discrete Mathematics, vol. 5, pp. 287–326. Elsevier (1979). https://doi.org/10.1016/S0167-5060(08)70356-X
14. Havlak, P.: Nesting of reducible and irreducible loops. ACM Trans. Program. Lang. Syst. **19**(4), 557–567 (1997). https://doi.org/10.1145/262004.262005
15. Ioannou, N., Kauschke, M., Gries, M., Cintra, M.: Phase-based application-driven hierarchical power management on the single-chip cloud computer. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 131–142 (2011). https://doi.org/10.1109/PACT.2011.19
16. Jackson, D.B., Snell, Q., Clement, M.J.: Core algorithms of the Maui scheduler. In: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP 2001, pp. 87–102. Springer, London, UK (2001). http://dl.acm.org/citation.cfm?id=646382.689682
17. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, pp. 85–103. Springer US, Boston, MA (1972). https://doi.org/10.1007/978-1-4684-2001-29
18. Khan, A.A., Mccreary, C.L., Jones, M.S.: A comparison of multiprocessor scheduling heuristics. In: Internatonal Conference on Parallel Processing Vol. 2, vol. 2, pp. 243–250 (1994). https://doi.org/10.1109/ICPP.1994.19
19. Lawler, E.L., Lenstra, J.K., Kan, A.H.R., Shmoys, D.B.: Chapter 9 sequencing and scheduling: Algorithms and complexity. In: Logistics of Production and Inventory, Handbooks in Operations Research and Management Science, vol. 4, pp. 445 – 522. Elsevier (1993). https://doi.org/10.1016/S0927-0507(05)80189-6
20. Lee, I., Iliopoulos, C.S., Park, K.: Linear time algorithm for the longest common repeat problem. J. Discret. Algorithms **5**(2), 243–249 (2007). https://doi.org/10.1016/j.jda.2006.03.019. 2004 Symposium on String Processing and Information Retrieval
21. Lenstra, J., Kan, A.R., Brucker, P.: Complexity of machine scheduling problems. In: Studies in Integer Programming, Annals of Discrete Mathematics, vol. 1, pp. 343–362. Elsevier (1977). https://doi.org/10.1016/S0167-5060(08)70743-X
22. Lopes, R.V., Menascé, D.: A taxonomy of job scheduling on distributed computing systems. IEEE Trans. Parallel Distrib. Syst. **27**(12), 3412–3428 (2016). https://doi.org/10.1109/TPDS.2016.2537821
23. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6), 529–543 (2001). https://doi.org/10.1109/71.932708
24. Ramalingam, G.: Identifying loops in almost linear time. ACM Trans. Program. Lang. Syst. **21**(2), 175–188 (1999). https://doi.org/10.1145/316686.316687

25. Rotithor, H.G.: Taxonomy of dynamic task scheduling schemes in distributed computing systems. IEE Proc. Comput. Digital Techn. **141**(1), 1–10 (1994). https://doi.org/10.1049/ip-cdt:19949630
26. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Selective reservation strategies for backfill job scheduling. In: Job Scheduling Strategies for Parallel Processing: 8th International Workshop, JSSPP 2002 Edinburgh, Scotland, UK, July 24, 2002 Revised Papers, pp. 55–71. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-36180-44
27. SuperMUC Petascale System (2017). https://www.lrz.de/services/compute/supermuc/. [Online]
28. Tarjan, R.: Testing flow graph reducibility. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC 1973, pp. 96–107. ACM, New York, NY, USA (1973). https://doi.org/10.1145/800125.804040
29. Transregional Research Center InvasIC (2017). http://www.invasic.de. [Online]
30. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). 10.1007/BF01206331
31. Ullman, J.: Np-complete scheduling problems. J. Comput. Syst. Sci. **10**(3), 384–393 (1975). https://doi.org/10.1016/S0022-0000(75)80008-0
32. Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings, pp. 170–183. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-211

Online Performance Analysis with the Vampir Tool Set

Matthias Weber, Johannes Ziegenbalg and Bert Wesarg

Abstract Today, performance analysis of parallel applications is mandatory to fully exploit the capabilities of modern HPC systems. Many performance analysis tools are available to support users in this challenging task. All tools usually employ one of two analysis methodologies. The majority of analysis tools, such as HPCToolkit or Vampir, follow a post-mortem analysis approach. In this approach, a measurement infrastructure records performance data during the application execution and flushes its data to the file system. The tools perform subsequent analysis steps after the application execution by using the stored performance data. Post-mortem analysis comes with the disadvantage that possibly large data volumes need to be handled by the I/O subsystem of the machine. Tools following an online analysis approach mitigate this disadvantage by avoiding the I/O subsystem. The measurement infrastructure of these tools uses the network to directly transfer the recorded performance data to the analysis components of the tool. This approach, however, comes with the limitation that the complete analysis occurs at application runtime. In this work we present a prototype implementation of Vampir capable of performing online analysis. We discuss advantages and disadvantages of both approaches and draw conclusions for designing an online performance analysis tool.

1 Introduction

Performance analysis and optimization has become essential for efficient usage of HPC resources. However, due to rising complexity in HPC software and hardware this is no trivial task. Often, the detection of performance bottlenecks in parallel applications requires performance analysis tools. As many types of performance problems only arise at larger scales, performance tools need to scale along with parallel applications [3, 8, 12]. This required tool scalability poses performance

M. Weber (✉) · J. Ziegenbalg · B. Wesarg
TU Dresden ZIH, Chemnitzer Str. 50, 01187 Dresden, Germany
e-mail: matthias.weber@tu-dresden.de

J. Ziegenbalg
e-mail: johannes.ziegenbalg@tu-dresden.de

C. Niethammer et al. (eds.), *Tools for High Performance Computing 2017*,
https://doi.org/10.1007/978-3-030-11987-4_8

challenges for the analysis tools themselves. One severe challenge with respect to scalability are the required I/O operations of performance data by the analysis tool. This is especially true in case of tracing tools, that record and retain very fine-grain application performance data. Such tools easily generate hundreds of gigabytes of performance data on current systems [7, 19]. While this fine-grain data is essential for the identification of various types of performance problems it also causes a substantial amount of I/O operations to file systems. Considering prevailing trends in HPC systems, such I/O operations at extreme scale might become unreasonable in the future [15].

Currently, two major types of performance analysis tool designs exist. So called *post-mortem* tools record application behavior at runtime and flush all recorded performance data to disk. The performance analysis itself is executed after the application run. The second type of performance tools perform *online* analysis. Such tools record and analyse performance data on-the-fly during the application run.

Compared to post-mortem tools, online tools provide the advantage of completely avoiding I/O operations to the file system. In this paper we discuss the design challenges when developing an online monitoring prototype with post-mortem tool components. We present an example measurement using our online prototype and evaluate the online approach.

Our contributions in this paper include:

- Analysis and discussion of the current measurement workflow.
- Design concept for an online-monitoring tool.
- Initial measurements using a prototype implementation of the online tool design.
- Discussion of the advantages of both common tool designs.

Our paper is organized as follows. We provide related work in Sect. 2 and follow with challenges for performance analysis tools in Sect. 3. Section 4 gives an overview of current tool analysis approaches, while Sect. 5 provides low-level design details of our prototype implementation. In Sect. 6 we present analysis results using our online monitoring prototype. Section 7 compares the online and post-mortem approach. Finally, in Sect. 8 we conclude and discuss future work.

2 Related Work

Many tools and approaches exist in the field of parallel performance analysis. Options range from basic tools dedicated to individual purposes, such as STAT [12], up to complex tools providing a rich feature set like Vampir [10].

The majority of performance analysis tools employ a post-mortem analysis approach. Examples include Arm MAP [1] and Vampir [10].

Besides the post-mortem approach, also online tools are used for performance analysis. Periscope [5] and Paradyn [13] are examples for this tool group. Periscope automatically detects wait states in parallel applications caused by inefficient communication behavior. Paradyn employs a dynamic instrumentation approach con-

trolled by a performance model to automate the identification of program parts that contribute significant execution time.

Moreover, several studies describe the design of scalable (online) analysis tools [2, 12, 16]. For scalable performance data aggregation many online tools use Tree Based Overlay Networks (TBON), like MRNet [14] in case of Paradyn.

Additionally to data processing capabilities, visual scalability is also important and challenging for parallel tools. Vampir uses timeline folding strategies [17] to maintain visual scalability. The tool Scalasca [18] generates aggregated performance reports out of large trace data sets in order to present summary information to the user.

3 Challenges for Extreme-Scale Performance Tools

Following we describe two major challenges we see in todays tracing-based performance analysis tools. Implementing an online-monitoring approach provides the potential to mitigate or overcome these challenges.

3.1 Challenge: Data Volume

One top issue connected to trace-based post-mortem analysis is always the loading and storing of performance data to/from the disk. The amount of measured performance data directly scales with the parallelism and duration of the target application. Eventually, the recorded performance data needs to be processed and stored. In extreme scale scenarios this poses a significant challenge to the complete analysis system, including hardware and software.

The following two real-world analysis examples demonstrate this situation in practice.

Ilsche et al. [7] describe a large-scale measurement run of the *S3D* application code (2, 3 PFLOP/s peak performance). S3D was measured running with 200,448 MPI processes on the HPC system Jaguar installed at ORNL. VampirTrace [10] recorded about 1 trillion performance events that resulted in about 5 TB compressed trace data in the OTF format [9]. Writing and analyzing 5 TB of performance data is already problematic and required 21,515 VampirServer analysis processes. On future machines, writing full-scale application traces to disk is not reasonable anymore and will probably overwhelm the I/O system.

Wylie et al. [19] describe a large-scale analysis of the *Sweep 3D* benchmark. Scalasca [18] was used to analyze the application running with 294,912 processes on a BG/P system. The initial application runtime was about 8 min. With enabled measurement infrastructure the runtime took about 9 min (measurement overhead of about 5%). The measurement generated 790 GB of performance data. The necessary creation of 294,912 files took about 86 min (original version) or 10 min (improved

version using SIONlib for aggregating multiple logical files into one physical file on disk). The unification of performance data took about 43 min (original version involving complete performance data rewrite) or 13 s (improved version). The Scalasca analysis replay step took about 11 s. The measured times in this example show that all activities related to I/O dominate the measurement.

This I/O challenge needs to be tackled in order to provide reasonable performance analysis solutions at extreme scale. The amount of data written to the file system to be needs to be kept at a minimum.

3.2 Challenge: Usability

An important success factor of performance analysis tools is their usability. Typically, developers are willing to invest only limited amounts of time in the performance analysis of their applications. Many developers quickly stop using tools that require too much effort for producing meaningful performance results.

Figure 1 shows the suggested workflow for an performance analysis using Score-P and Vampir. The most critical step in this workflow is the recompilation/relinking of the application. While straightforward for some applications, this step can become literally impossible for others. As build systems of scientific application tend to be involving, rebuilding the application with the measurement system can become a serious challenge. The second possibly difficult step is the creation of an appropriate filter file in order to keep the measurement overhead at a reasonable level. While not even necessary for some applications, this step can also become cumbersome for

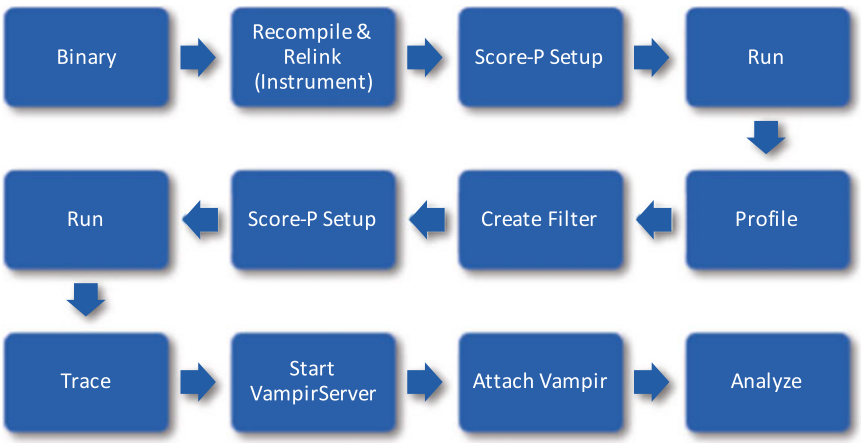


Fig. 1 Suggested workflow of a Score-P/Vampir performance analysis

Fig. 2 Performance analysis workflow providing good usability

many application types. Especially the creation of filter files for C++ applications might become involving.

Figure 2 depicts the ideal performance analysis workflow. In this scenario developers can directly use their application binaries and start with the measurement. Typically, they start the performance analysis tool together with the application binary. No rebuild of the application or creation of filtering files is needed. During the application run, the analysis tool records performance data and ensures low measurement overhead. In case of our online analysis prototype, the developer sees performance data directly as the application executes. In this scenario the initial effort for generating meaningful performance data is kept to a minimum.

4 Prevailing Performance Tool Designs

Most parallel performance analysis tools employ one of the following two design approaches. Both designs can implement profiling as well as tracing analysis techniques.

4.1 Post-Mortem Approach

The majority of parallel performance analysis tools use this design. Key characteristic is that the actual performance analysis is executed after the application run has finished. Therefore, measured performance data has to be retained in the file system after the application run.

The Vampir/Score-P analysis tool set, Fig. 3, provides an example of this design. The measurement system Score-P [11] is attached to the application and records performance data during the application execution. After the application has finished, Score-P writes all measured performance data to disk using the OTF2 [4] file format. The actual performance analysis is executed with Vampir [10], reading the OTF2 files from disk back into memory. The parallel VampirServer component allows to load and analyze performance data volumes of large-scale application runs.

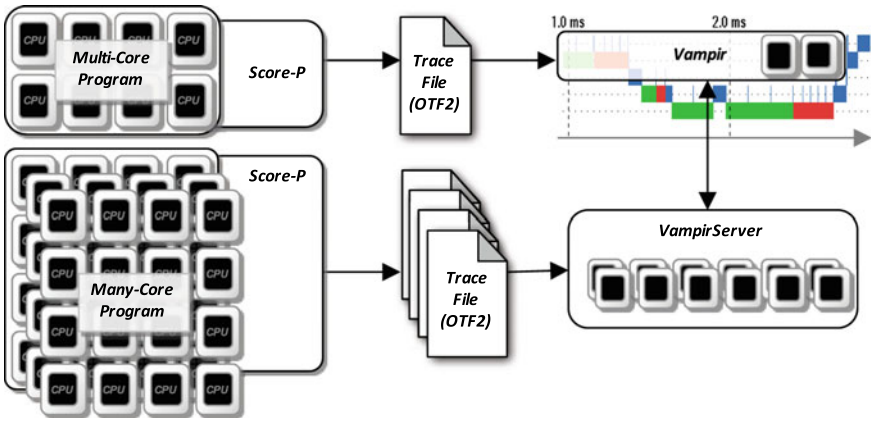


Fig. 3 Post-mortem tool design. All performance data is captured during the application run (Score-P) and written to the file system (OTF2 files). Analysis of the performance data is executed after the application run (Vampir)

Fig. 4 Legend for Figs. 5, 6, and 7

4.2 Online Approach

The second approach implements online analysis tools. Key characteristic of this design is that the performance analysis is performed during the application execution. Figures 4 and 5 provide an overview of this design. For recording performance data a measurement infrastructure is attached to the target application. Parallel to the application run dedicated analysis tool resources. The measurement infrastructure uses the network to directly transfer measured data to the analysis processes. To maintain scalability, many tools like STAT [12] and Paradyn [13] employ an Tree Based Overlay Networks (TBON) component. This tree layout allows to scalably aggregate performance data. The root node of the tree typically generates a performance report or continuously updates displayed performance data.

5 Building an Online-Monitoring Tool

To overcome the challenges described in Sect. 3 we designed an online analysis prototype. In this section we share our experiences in designing an online tool using existing post-mortem components.

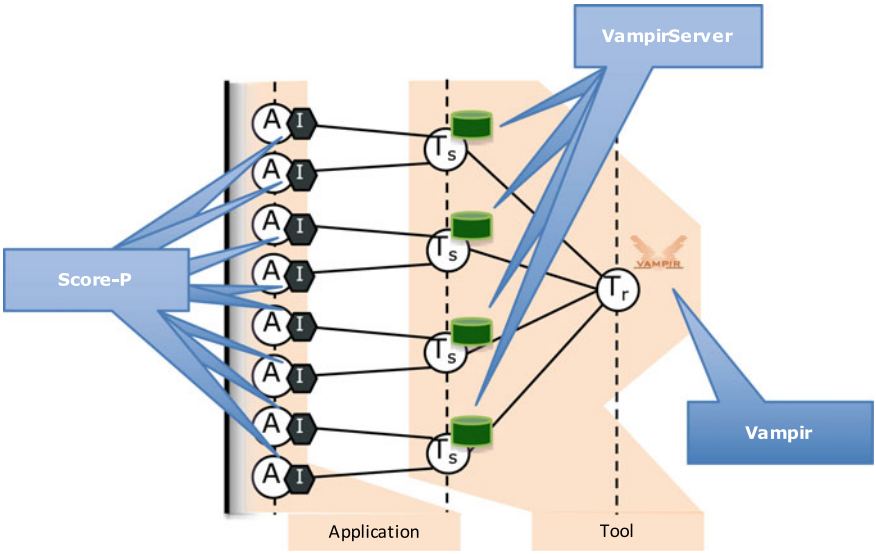


Fig. 5 Online-monitoring architecture overview showing individual tool components

5.1 Going from Post-Mortem to Online

The Vampir analysis tool set already provides basic components required for an online analysis tool. Like shown in Fig. 5 we use the recording component Score-P [11] and the analysis components Vampir and VampirServer [10] for our prototype. We measure performance data using sampling. This allows to measure uninstrumented application binaries and to avoid the recompilation step.

Following the online design described in Sect. 4 all tool components need to run parallel to the application. Therefore, we extend the usual target application resource allocation to cover the additional resources for the analysis components. At application startup (with Score-P attached) we also launch all required VampirServer processes. Additionally, we establish communication paths from Score-P processes to corresponding VampirServer processes. Score-P uses these paths throughout the application run to transfer performance data to the VampirServer processes. All performance data is stored in the main memory of the receiving VampirServer processes. This procedure is indicated in Fig. 6.

Finally, we launch a Vampir instance that connects to VampirServer. Vampir displays the recorded performance data to the user. In order to provide online monitoring information Vampir continuously issues requests for new performance data. The VampirServer processes holding the performance data compute the issued visualization requests and send performance information to the Vampir client, Fig. 7. However, the computation of these requests is time critical in an online scenario.

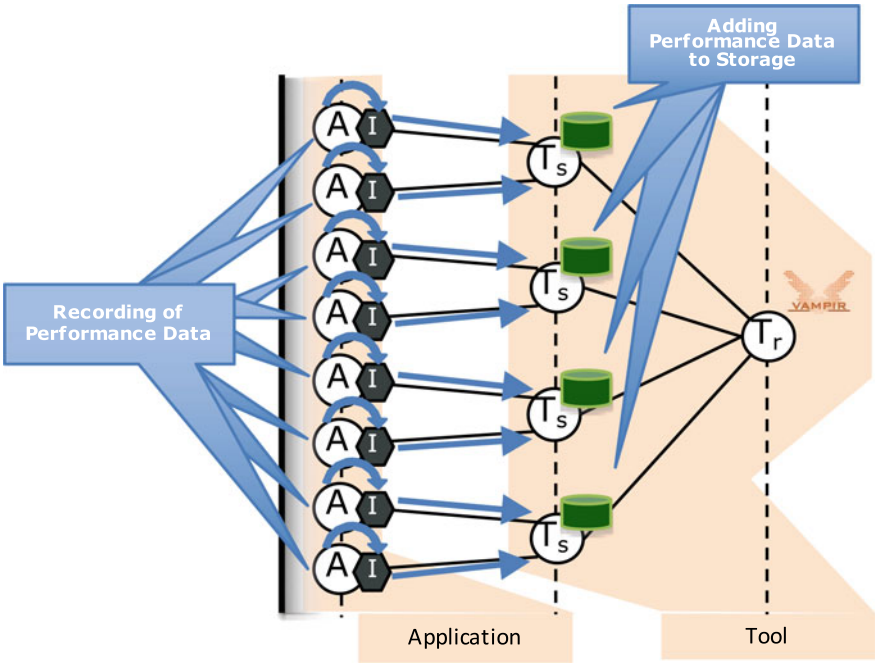


Fig. 6 Recording, transfer and storage of performance data

To provide the user with fluent continuous visual updates of the performance data, Vampir frequently issues visualization requests in the range of about 500 ms. The VampirServer processes need to be capable to answer these requests in time in order to prevent overloading. Additionally to the computation of visualization requests, the VampirServer processes also need to receive and process incoming performance data at the same time, Fig. 6.

An important factor to control the load on VampirServer processes is their fan-in, i.e. the number of Score-P processes that connect to one VampirServer process. The fan-in in Fig. 5 is 2. In practice we found a fan-in of 128 (128 application/Score-P processes connect to one VampirServer process) working well. This number keeps load on VampirServer processes at reasonable level and avoids overloading while limiting required analysis tool resources at the same time.

The common displays of Vampir are not designed to visualize continuous performance data updates. In order to provide performance information of a running application we developed a new monitoring view. This new view, Fig. 8, shows only aggregated performance information and provides therefore great scalability potential. Besides summarized profile information, the view also shows statistical information over time. The view is designed to handle continuously incoming new performance data, that is appended at the right side of the timeline view. New data

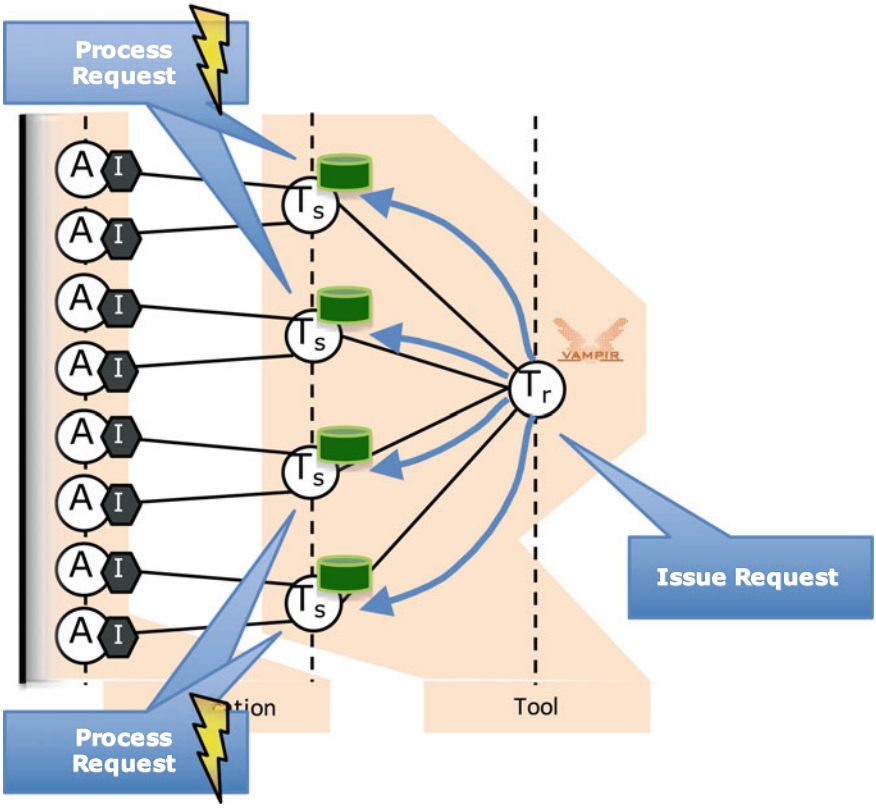


Fig. 7 Time-critical processing of visualization requests

also automatically updates all profile information. Using the monitoring view allows users to visually inspect the performance of running applications.

Together, all components build our online analysis prototype. This tool shows that it is possible to design an online analysis tool using post-mortem components. However, due to the nature of online performance analysis, a couple of adaptions are necessary. We discuss these in the next section.

5.2 Design Considerations for an Online-Monitoring Tool

The following list summarizes design and implementation challenges that require adaption or extension of existing post-mortem tool components.

Extra system resources required. Besides the usual resources required for the application execution, an online tool additionally requires dedicated system resources for processing performance data.

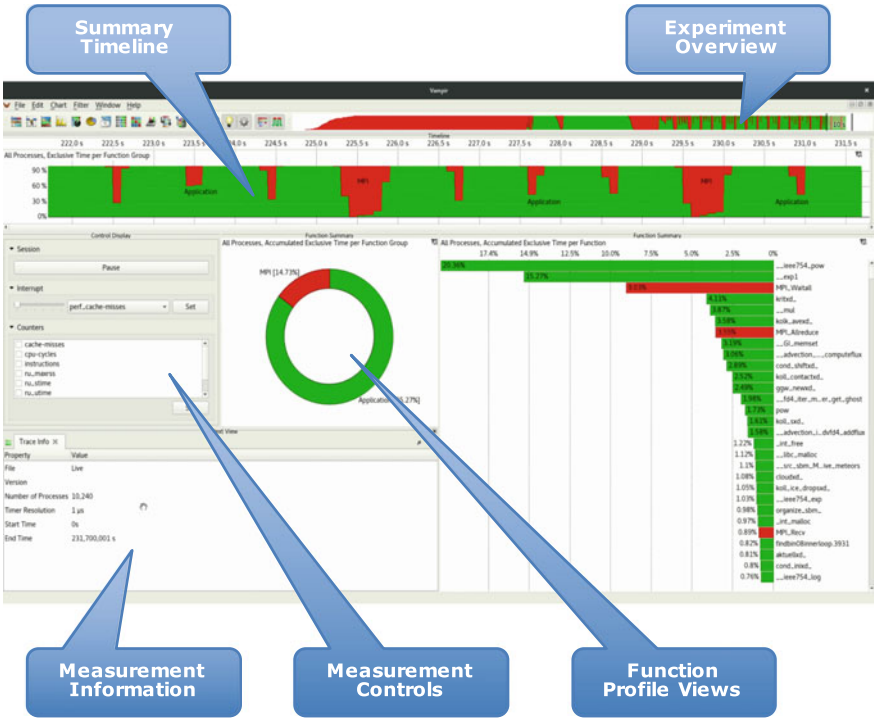


Fig. 8 The new monitoring view of Vampir showing performance information of a running application

Available main memory for performance data limits recording capabilities. As performance data is not flushed to disk, the tool retains all recorded data in main memory. Due to dedicated resources the tool does not interfere with application memory. However, available tool memory dictates possible measurement time.

Complex application startup required in online scenario. All tool components need to be started together with the application. Additionally, the online tool requires initiation of communication paths between the recording and the processing tool components.

Additional load on network due to performance data transfer. Recording tool components may influence application communication behavior as they send performance data over the network.

Processing speed of performance data is critical in online scenario. Processing tool components need to handle new incoming performance data and answer visualization requests at the same time. This renders expensive and time-consuming operations prohibitive. As performance data is generated continuously, tool components need to keep up with this data stream for providing a monitoring view of the performance data.

Limited analyses capabilities in online scenario. Analyses that require complete data sets are impossible. As performance data is continuously appended and potentially incomplete, some analyses, such as critical-path analysis or message matching, are not feasible.

Aggregated visualizations are suitable in online scenario. As processing capabilities of performance data is critical, aggregated data and visualizations provide the most scalability potential. Non-scalable visualizations easily violate processing requirements of an online tool.

Monitoring visualization view required in online scenario. To provide the user with continuous performance data updates, a dedicated view is necessary. Existing post-mortem visualizations are impractical for displaying monitoring information.

Continuous time synchronization required for longer application runs. Clocks on many HPC systems drift and continuously change their speeds, seriously deteriorating measurement accuracy. Strategies employed by post-mortem tools are not applicable in an online scenario. Online tools need to implement continuous time synchronization methods to maintain high measurement accuracy.

On-the-fly unification of data required. Post-mortem tools perform the necessary unification of performance data after the application execution. This is not possible in an online scenario. Online tools need to unify recorded performance metadata on-the-fly during the running measurement.

Attending the measurement is required for performance analysis. As the online tool does not retain any performance data, the user needs to attend the measurement to perform the analysis. Since especially large-scale jobs may start at unsuitable times for the user, this restriction poses a serious usability limitation.

6 Online Performance Analysis Showcase–COSMO-SPECS

In this section we demonstrate an example measurement using our online analysis prototype. The case study evaluates an analysis run of the weather forecast code COSMO-SPECS [6]. This application couples two models, *COSMO* and *SPECS*, for more accurate simulations of cloud and precipitation processes. The COSMO regional weather forecast model was developed at the German Weather Service (DWD). The SPECS cloud microphysics model was developed at the Leibniz Institute for Tropospheric Research (IfT). SPECS computes detailed interactions between aerosols, clouds, and precipitation.

In order to compare our prototype with the current post-mortem Score-P/Vampir solution, we present such a reference measurement first. We had to rebuild COSMO-SPECS for inserting the instrumentation instructions and attaching Score-P. The following characteristics describe a Score-P measurement run of COSMO-SPECS with full instrumentation enabled.

Reference Measurement Using Score-P/Vampir	
Measurement method:	Full Instrumentation
Application parallelism:	64 cores
Application runtime:	1 min 27 s
Recorded performance data:	35 GiB ($6.4 \frac{MiB}{process} \frac{1}{s}$)

In such a scenario the measurement system records performance data in a rate of about 6.4 MiB/s for each process. In about 1.5 min 64 processes produced already 35 GiB performance data. It is obvious that such high measurement data rates cause severe perturbation to the original application run, rendering the measurement unusable and stressing the entire I/O subsystem considerably. To be able to measure higher process numbers and to retrieve usable measurement data involves filtering steps according to the workflow described in Sect. 3.2. Simply starting with a fully instrumented binary is not sufficient for an initial analysis of the COSMO-SPECS performance.

Following we measure the same COSMO-SPECS application using our online prototype. For this measurement we did not prepare the original application binary in any way.

Vampir Online Prototype Measurement Run	
Measurement method:	Sampling (250 Hz)
Application parallelism:	10,240 cores
Tool processes:	86
Application runtime:	14 min 56 s
Recorded performance data:	71 GiB ($8.1 \frac{KiB}{process} \frac{1}{s}$)
Runtime overhead:	$<5\%$
Resource overhead:	$<1\%$ (86 : 10,240)

Figure 8 shows measured performance data of the COSMO-SPECS measurement run. Our prototype employs a sampling measurement approach. This keeps the performance data rate low at about 8.1 KiB/s for each process. Consequently, it is possible to record much higher process numbers (10,240) for longer durations (15 min). The total amount of recorded performance data resides distributed over the tool processes. This significantly eases memory requirement for individual tool processes. Additionally, it is possible to scale tool resources as necessary with the application. Overall, the resource overhead induced by the online approach is negligible at below one percent. Our measurement caused a runtime overhead below five percent. This shows that without prior filtering our prototype can provide meaningful performance data using unprepared application binaries.

7 Comparison of the Online with the Post-Mortem Approach

This section compares the online approach with the established post-mortem approach.

The online approach completely avoids I/O operations to the file system. This allows to overcome the severe scalability limitations presented Sect. 3.1 and in turn suggests high scalability potential of the online approach. Additionally, the online approach enables more flexible measurement control during application runtime. Especially when actively supervising a measurement, an user can directly control and alter measurement parameters, such as activating/deactivating the measurement of particular performance counters or changing the sampling frequency. As explained in Sect. 3.2, our online analysis prototype provides an easier measurement workflow compared to the established post-mortem workflow. This is particularly valuable for applications that are relatively unknown to the analyst. In such cases the user can completely avoid the initial measurement preparation (including rebuild, filtering, etc.) required by the post mortem approach. However, this advantage also imposes limitations on the analysis capabilities. The post-mortem approach provides more options for detailed in-depth analysis. The online approach comes with the disadvantage of requiring additional resources at application runtime. This complicates the application job setup for the analyst as well as introduces a more complex job startup on the tool side. Finally, the online approach requires users to always be able to actually observe the measurement run. Especially considering large-scale application runs, this is a rather unrealistic assumption.

8 Conclusions

In this paper we present design concepts and an initial prototype for analyzing application performance in an online approach. Further, we compare the online-monitoring design with the established post-mortem analysis design. The introduced online design solves I/O problems due to overloading of file systems. This presents scalability potential for large-scale performance analysis tools.

The permanent storage of performance data for later analyses and reference is also desirable in the online case. This should be considered in the design of an online analysis tool.

We can conclude that both approaches are useful. The online approach is especially useful for initial analyses of unknown code and large-scale application runs. The post-mortem approach is suitable for very detailed analyses, especially if the application is well know to the analyst.

In the future we plan to follow the described design ideas and further improve our prototype.

References

1. Arm Forge (Arm MAP) Version 18.0 (2017). https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge
2. Brunst, H., Malony, A.D., Shende, S.S., Bell, R.: Online remote trace analysis of parallel applications on high-performance clusters. In: Veidenbaum, A., Joe, K., Amano, H., Aiso, H. (eds.) High Performance Computing: 5th International Symposium, ISHPC 2003, Tokyo-Odaiba, Japan, October 20–22, 2003. Proceedings 13, pp. 440–449. Springer, Berlin, Heidelberg (2003)
3. Brunst, H., Weber, M.: Custom hot spot analysis of HPC software with the Vampir performance tool suite. In: Proceedings of the 6th International Parallel Tools Workshop, pp. 95–114. Springer, Berlin, Heidelberg, September 2012
4. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W., Wolf, F.: Open trace format 2: the next generation of scalable trace formats and support libraries. In: Proceedings of the 14th Biennial ParCo Conference, vol. 22 of ParCo2011, pp. 481–490, January 2012
5. Gerndt, M., Ott, M.: Automatic performance analysis with periscope. Concurr. Comput. Pract. Expe. **22**(6), 736–748 (2010)
6. Grützun, V., Knoth, O., Simmel, M.: Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM-SPECS: model description and first results. Atmos. Res. **90**(24), 233–242 (2008)
7. Ilsche, T., Schuchart, J., Cope, J., Kimpe, D., Jones, T., Knüpfer, A., Iskra, K., Ross, R., Nagel, W.E., Poole, S.: Enabling event tracing at leadership-class scale through I/O forwarding middleware. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2012, pp. 49–60. ACM, New York, NY, USA (2012)
8. Kitayama, I., Wylie, B.J.N., Maeda, T.: Execution performance analysis of the ABySS genome sequence assembler using Scalasca on the K computer. In: Parallel Computing: On the Road to Exascale, volume 27 of Advances in Parallel Computing, pp. 63–72. International Conference on Parallel Computing 2015, Edinburgh (Scotland), 1 Sep 2015–4 Sep 2015, IOS Press, September 2016
9. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the open trace format (OTF). In: Proceedings of the 6th International Conference on Computational Science - Volume Part II, ICCS 2006, pp. 526–533. Springer, Berlin, Heidelberg (2006)
10. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.), Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing. Springer, Berlin, Heidelberg, July 2008
11. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In: Proceedings of 5th Parallel Tools Workshop, pp. 79–91. Springer, Berlin, Heidelberg (2012)
12. Lee, G.L., Ahn, D.H., Arnold, D.C., de Supinski, B.R., Legendre, M., Miller, B.P., Schulz, M., Liblit, B.: Lessons learned at 208K: towards debugging millions of cores. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 26:1–26:9. IEEE Press, Piscataway, NJ, USA, (2008)
13. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. Computer **28**(11), 37–46 (1995)
14. Roth, P.C., Arnold, D.C., Miller, B.P.: MRNet: a software-based multicast/reduction network for scalable tools. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC 2003. ACM, New York, NY, USA (2003)
15. TOP500 List of the World's Fastest Supercomputers (2017). http://www.top500.org

16. Wagner, M., Hilbrich, T., Brunst, H.: Online performance analysis: an event-based workflow design towards Exascale. In: 2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and Systems (HPCC, CSS, ICESS), pp. 839–846, August 2014
17. Weber, M., Geisler, R., Brunst, H., Nagel, W.E.: Folding methods for event timelines in performance analysis. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 205–214. IEEE Computer Society, May 2015
18. Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Frings, W., Fürlinger, K., Geimer, M., Hermanns, M.-A., Mohr, B., Moore, S., Pfeifer, M., Szebenyi, Z.: Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In: Proceedings of the 2nd Parallel Tools Workshop, Stuttgart, Germany, pp. 157–167. Springer, July 2008
19. Wylie, B.J.N., Geimer, M., Mohr, B., Böhme, D., Szebenyi, Z., Wolf, F.: Large-scale performance analysis of Sweep3D with the Scalasca toolset. Parallel Proces. Lett. **20**(4), 397–414 (2010)