João M. Lourenço
Eitan Farchi (Eds.)

LNCS 8063

Multicore Software Engineering, Performance, and Tools

International Conference, MUSEPAT 2013
St. Petersburg, Russia, August 2013
Proceedings

Springer

Lecture Notes in Computer Science 8063

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

João M. Lourenço Eitan Farchi (Eds.)

Multicore Software Engineering, Performance, and Tools

International Conference, MUSEPAT 2013
St. Petersburg, Russia, August 19-20, 2013
Proceedings

Springer

Volume Editors

João M. Lourenço
Universidade Nova de Lisboa, Departamento de Informática FCT-UNL
2829-516 Caparica, Portugal
E-mail: joao.lourenco@fct.unl.pt

Eitan Farchi
Haifa University, IBM Haifa Research Laboratory
3190501 Haifa, Israel
E-mail: farchi@il.ibm.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39954-1 e-ISBN 978-3-642-39955-8
DOI 10.1007/978-3-642-39955-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013943830

CR Subject Classification (1998): D.3.3-4, D.2.11, D.1.3, C.1.4, D.2.2, C.3-4
D.2, D.1.5, D.4.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

It is our pleasure to welcome you to the proceedings of the International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT 2013)!

MUSEPAT merges and brings together the communities of the International Workshops on Multicore Software Engineering (IWMSE) and on Parallel and Distributed Systems: Testing, Analysis and Debugging (PADTAD). MUSEPAT strives to provide a venue where researchers from academia and industry interested in the challenges of multicore systems can present their latest scientific contributions and participate in open discussions. MUSEPAT 2013 continued this tradition and included two keynote addresses, one from the industry and the other from academia, three technical sessions, and a brainstorming session.

This year's keynote speakers were Dr. Zakhar A. Matveev, from Intel, Russia, and Prof. Nuno Preguiça, from Universidade Nova de Lisboa, Portugal. The call for papers attracted 25 submissions by 76 authors from all over the world. All papers were rigorously peer reviewed by at least three members of the Program Committee. Nine high-quality papers were accepted to the conference and organized into three main sessions: Performance Analysis and Algorithms, Programming Models and Optimization, and Testing and Debugging. The brainstorming session is an open time for small group discussions and informal presentations that allow conference participants to discuss and share new ideas and challenges in software engineering for multicore systems.

We would like to thank our keynote speakers and all authors for sharing their research contributions with us and with the larger research community. We would also like to thank the members of the Program Committee and external reviewers for their help in evaluating and selecting high-quality papers, as well as Springer for their support with publishing the proceedings in the LNCS series. We are thankful to the MUSEPAT Steering Committee members for their guidance and support during the planning and organization of this conference. Special thanks also go to Bertrand Meyer, General Chair of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013), and the rest of the ESEC/FSE Organizing Committee for hosting MUESPAT as a co-located conference. Our final thanks go to our corporate sponsors, Intel and IBM Research, who generously support MUSEPAT 2013.

On behalf of the MSUEPAT 2013 Program Committee, we hope that you find this year's proceedings interesting and insightful.

August 2013
João M. Lourenço
Eitan Farchi

Organization

Organizing Committee

Conference Chair

João M. Lourenço

Program Chair

Eitan Farchi

Program Committee

Jeremy Bradbury	University of Ontario, Institute of Technology, Canada
Eitan Farchi	IBM Haifa Research Laboratory, Israel
Klaus Havelund	NASA's Jet Propulsion Laboratory, USA
Michael Hind	IBM Thomas J. Watson Research Center, USA
Akash Lal	Microsoft Research, India
João Lourenço	Universidade Nova de Lisboa, Portugal
Shiva Nejati	University of Luxembourg, Luxembourg
John Owens	University of California-Davis, USA
Victor Pankratius	MIT, USA
Paul Petersen	Intel, USA
Michael Philippsen	University of Erlangen-Nuremberg, Germany
Christian Prehofer	LMU München, Germany
Scott D. Stoller	Stony Brook University, USA
Tomas Vojnar	Brno University of Technology, Czech Republic
Shmuel Ur	University of Bristol, UK

Steering Committee

Jeremy Bradbury	University of Ontario, Institute of Technology, Canada
Eitan Farchi	IBM Haifa Research Laboratory, Israel
João Lourenço	Universidade Nova de Lisboa, Portugal
Victor Pankratius	MIT, USA
Michael Philippsen	University of Erlangen-Nuremberg, Germany
Christian Prehofer	LMU München, Germany
Shmuel Ur	University of Bristol, UK

External Reviewers

Vitor Duarte	Universidade Nova de Lisboa, Portugal
Jan Fiedor	Brno University of Technology, Czech Republic
Vendula Hruba	Brno University of Technology, Czech Republic
Bohuslav Krena	Brno University of Technology, Czech Republic
João Leitão	Universidade Nova de Lisboa, Portugal
Zdenek Letko	Brno University of Technology, Czech Republic
Pedro Medeiros	Universidade Nova de Lisboa, Portugal
Yu Qi	Mesh Capital LLC, USA
Shmuel Ur	University of Bristol, UK
Tiago M. Vale	Universidade Nova de Lisboa, Portugal

Table of Contents

Performance Analysis and Algorithms

Programming Models and Optimization

Testing and Debugging

Self-timed Scheduling and Execution of Nonlinear Pipelines with Parallel Stages

Lars Lucas[1], Tobias Schuele[2], and Wolfgang Schwitzer[1]

[1] Technische Universität München, Fakultät für Informatik
Boltzmannstr. 3, 85748 Garching, Germany
{lucas,schwitze}@in.tum.de
[2] Siemens AG, Corporate Technology
Otto-Hahn-Ring 6, 81739 München, Germany
tobias.schuele@siemens.com

Abstract. Applications that process continuous streams of data, e.g., sensor signals, video images, network packets, etc. are well-suited for pipelined execution on multicore processors. In many cases, however, the applications are subject to real-time constraints, especially in embedded systems. Besides maximizing the throughput, it is therefore important to minimize deviations in the timing. To solve this problem, we propose a method for self-timed scheduling and parallel execution of stream-based applications in soft real-time environments. Our experimental results show significantly lower latencies compared to state-of-the-art approaches, while achieving high throughput.

1 Introduction

Multicore processors, which are prevalent in laptops, desktop computers, and servers, increasingly find their way into embedded systems [1]. For instance, many smartphones already contain processors with two or more cores. Unlike personal computers, however, embedded systems are often subject to real-time constraints. Typical examples are vehicle controllers, medical devices, and machines for industrial automation. In such systems, predictability of the timing plays an important role to ensure correct interaction with the environment. In general, one distinguishes between soft and hard real-time systems. For the former it must be guaranteed that the given deadlines are met in all cases. Hard real-time systems are typically found in safety-critical areas like aviation, where high demands are put on reliability. In soft real-time systems, a deadline may be missed without causing harm. However, to maintain the quality of service, a major goal in the design of such systems is to reduce deviations from the desired timing.

Consider, for example, a sorting machine that discards broken objects (Fig. 1). The machine consists of a conveyor belt, which transports the objects to be sorted at a fixed speed, a camera, which produces a video stream of the objects on the conveyor belt, and a pusher, which pushes broken objects into a box. The camera and the pusher are connected to a computer, which analyzes the incoming video stream and triggers the pusher when a broken object is detected or the considered object could not be classified within the given time span. In order to run the conveyor belt at a high speed, the image processing application must have a high throughput. Additionally, the latency should be

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 1–12, 2013.

as low as possible to keep the conveyor belt as short as possible. Increasing the distance between the camera and the pusher relaxes the timing constraints, but increases the costs and requires additional space. As usual in soft real-time systems, the deadline may be missed without causing harm. Nevertheless, this should only happen in rare cases to reduce the number of intact objects that are thrown away.

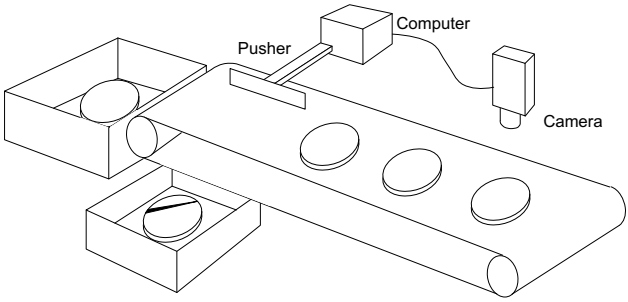


Fig. 1. Example for a soft real-time system (sorting machine)

In the past decades, a plethora of techniques for scheduling tasks on parallel systems have been developed, ranging from completely static to fully dynamic approaches [2]. Static scheduling algorithms are used to compute schedules before an application is executed [3,4,5]. Since the points of time at which the tasks are executed are known a priori, static scheduling allows a maximum degree of predictability. However, static scheduling algorithms require precise information on the worst-case execution times of the tasks, which are hard to obtain for modern multicore processors with shared caches. In fully dynamic approaches, all scheduling decisions are performed at run-time. A prominent representative of this class is work stealing, where idle processor cores steal tasks from other (busy) cores to balance the load [6]. A major advantage of dynamic scheduling is the ability to deal with varying execution times that may arise when shared resources such as the main memory are accessed. The downside is limited predictability, since there is usually little a priori information on the timing.

Self-timed scheduling approaches combine the advantages of both worlds. This is achieved by statically assigning the tasks to the processor cores using (not necessarily safe) estimates of the tasks' worst-case execution times. The result is a list for each processor core specifying the sequential ordering of the tasks. As in static scheduling, the lists are processed in an infinite loop, but the points of time at which the tasks are executed are determined dynamically. To this end, the runtime systems keeps track of the available data and ensures that a task is not executed until all input data is available, i.e., synchronization is performed during run-time. This makes self-timed scheduling robust with respect to changes in the execution times of the tasks.

To utilize the power of multicore processors, computations have to be split into tasks that can be executed in parallel. For that purpose, applications that process continuous streams of data, such as the sorting machine described previously, may be executed in a pipelined fashion [7,8]. This is one of the reasons why stream processing, which has a long history in computer science [9], has gained resurgent interest in recent years

(see, e.g., [10,11,12]). In its basic form, however, the throughput of a pipeline is limited by the slowest stage. To solve this problem, multiple invocations of a stage may be executed in parallel [13]. Hence, a new item, subsequently also called token, may enter a parallel stage before the previous one has left it, provided that the invocations do not interfere. For stages with side effects it is usually required that at most one invocation is active at a time. Such stages are said to be serial.

Chain-like (linear) pipelines increase the throughput, but do not decrease the latency, the time required to process an element from a stream. A reduction of the latency can be achieved using nonlinear pipelines, which may exhibit an arbitrary structure. In addition to temporal parallelism, such pipelines exploit spatial parallelism by processing independent substreams in parallel. In digital signal processing, for example, streams are frequently fed into multiple filter chains that may operate simultaneously.

In this paper, we present a method for self-timed scheduling of nonlinear pipelines with parallel stages. Our approach is well-suited for stream processing applications in soft real-time systems, especially when latency is a crucial concern. It allows developers of embedded systems to leverage from multicore processors without the need for complex dynamic schedulers, which often exhibit unpredictable runtime behavior.

An efficient algorithm for scheduling pipelines on multiprocessor systems was presented in [14]. In contrast to our approach, however, it cannot deal with nonlinear pipelines. The scheduling algorithm described in [15] solves this problem, but is optimized for throughput and does not aim to minimize the latency. The work of Banerjee et al. [16] has the same objective, but also supports heterogeneous systems. All three approaches assume a fixed assignment of pipeline stages to processors (cores), which may significantly hurt cache locality, since data flowing between successive pipeline stages must be physically transferred between the corresponding cores. Our approach avoids this problem in that successive stages are preferably scheduled on the same core.

The remainder of this paper is organized as follows: In the next section, we describe the foundations of our work and introduce the notion of extended task graphs used to model stream processing applications. In Sect. 3, we present our scheduling algorithm, which is based on the modified critical path method [17]. After that, we sketch the implementation (Sect. 4) and discuss experimental results (Sect. 5). Finally, we conclude with a summary and directions for future work in Sect. 6.

2 Foundations

An extended task graph (ETG) consists of a set of nodes that represent the computations of an application and a set of edges that represent the data flow. ETGs are essentially ordinary task graphs [4] except that each node has an additional attribute that specifies whether it is serial or parallel.

Definition 1. *An extended task graph is a directed acyclic graph* $G = (V, E, r, p, c)$*, where*

- V *is the set of nodes,*
- $E \subset V \times V$ *is the set of edges,*
- $r : V \to \mathbb{N}^+$ *is a function that associates with each node its computations costs,*

- $p : V \to \{0, 1\}$ *is a function that returns* 1 *if the node is parallel, and*
- $c : E \to \mathbb{N}^+$ *is a function that associates with each edge* $(v, w) \in E$ *the communication costs between nodes* v *and* w.

A node without predecessors is a *source* and a node without successors is a *sink* (for simplicity, we restrict ourselves to ETGs with a single source and a single sink). We write $v \rightsquigarrow w$ iff there is a path from v to w with $v, w \in V$ ($v \rightsquigarrow v$ holds for all $v \in V$). Additionally, we need the following definitions (see [4]):

- The *length* of a path $p = [v_i, v_{i+1}, \ldots, v_j]$ is the sum of the computation and communication costs, i.e., $\mathrm{len}(p) = \sum_{i \leq k \leq j} r(v_k) + \sum_{i \leq k < j} c(v_k, v_{k+1})$.
- The *bottom level* $\mathrm{bl}(v)$ of a node v is the length of the longest path from v to the sink v_{sink}, i.e., $\mathrm{bl}(v) = \max\{len(p) \mid p = [v, \ldots, v_{sink}]\}$.[1]
- A *critical path* cp is a longest path, i.e., $\mathrm{len}(\mathrm{cp}) = \mathrm{bl}(v_{source})$.
- The *as-late-as-possible* (ALAP) time of a node v is defined as $\mathrm{alap}(v) = \mathrm{len}(\mathrm{cp}) - \mathrm{bl}(v)$.

Intuitively, the ALAP time is the latest point of time a node must be scheduled without unnecessarily increasing the span of the resulting schedule. Figure 2 shows a sample ETG consisting of five nodes, where multiple instances of nodes C and D may be executed in parallel. The length of the critical path $[A, C, D, E]$ is 14. The bottom levels and ALAP times are given next to each node.

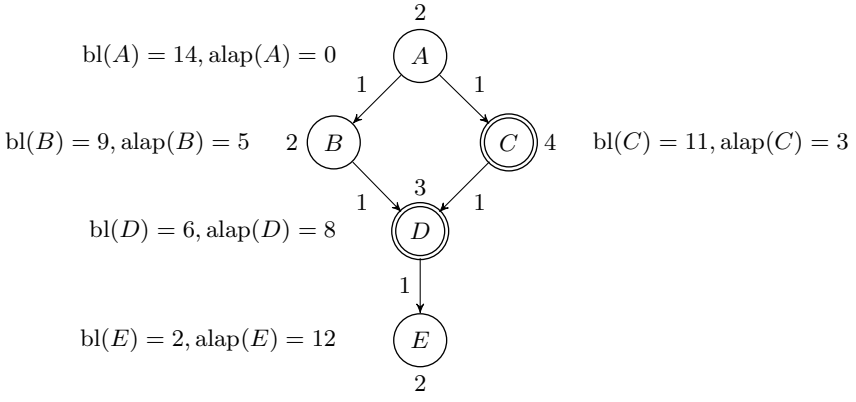


Fig. 2. Sample ETG consisting of five nodes (nodes C and D are parallel)

In order to exploit pipeline parallelism, we unfold ETGs. For that purpose, we create a fixed number of copies of an ETG, where each copy represents a run through the pipeline. Recall, however, that serial stages must be executed one after the other. This can be accomplished by introducing additional edges in the unfolded ETG such that the copies of a serial stage depend on each other. More formally, we denote by $G_i = (V_i, E_i, w_i, p_i, c_i)$ the ETG obtained from an ETG $G = (V, E, r, p, c)$ by labeling every

[1] An algorithm for computing bottom levels can be found in [4].

node $v \in V$ with index i such that G and G_i are isomorphic. The union of two ETGs G_i and G_j, written $G_i \cup G_j$, is defined elementwise.

Definition 2. *Let $(V, E, w, p, c) = \bigcup_{i=1}^{n} G_i$ be the ETG consisting of n copies of an ETG G. Then, $G' = (V, E \cup E', r, p, c')$ is the ETG obtained by unfolding G n times, where $E' \subset V \times V$ and $c' : E \cup E' \to \mathbb{N}$ are defined as follows:*

$$E' = \{(v_i, v_{i+1}) \mid \neg p(v_i) \wedge 1 \leq i < n\}$$

$$c'(e) = \begin{cases} 0 & \text{if } e \in E' \\ c(e) & \text{otherwise} \end{cases}$$

The above definition ensures that for every serial node there is an edge between successive runs. This way, parallel execution of different instantiations of a serial node is avoided. Since these additional edges are used for synchronization only and not for exchanging data, we set their communication costs to zero. Figure 3 shows the ETG of Fig. 2 unfolded three times (the computation and communications costs of the original ETG are omitted for the sake of clarity).

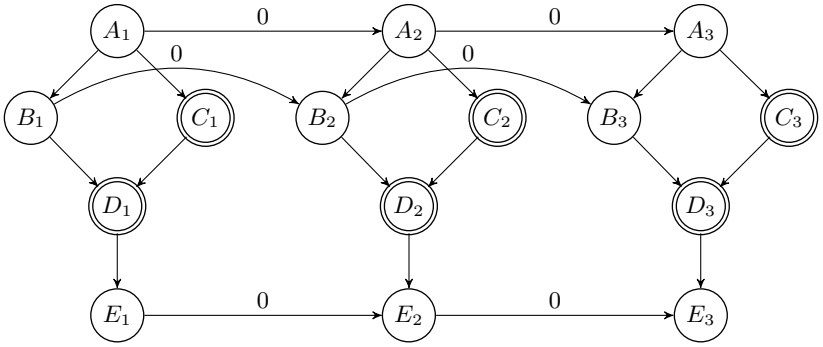


Fig. 3. ETG of Fig. 2 unfolded three times

3 Scheduling Algorithm

Having unfolded an ETG, we can, at least in principle, employ any static scheduling algorithm to obtain a schedule. It is not even necessary to distinguish between serial and parallel stages, as this information is implicitly encoded in the synchronization edges. Since we consider self-timed scheduling, the temporal assignment of tasks to processors can be ignored. In fact, we are only interested in the processor allocation, i.e., the sequence of tasks executed by a processor. As mentioned in the introduction, synchronization is performed on-the-fly to cope with varying runtimes. Note, however, that the resulting schedules only comprise finitely many runs. In order to execute nonterminating applications, the schedules are repeated in an infinite loop. This is referred to as blocked scheduling, where the number of unfoldings is called the blocking factor [1].

Most scheduling algorithms aim to minimize the span of the resulting scheduling. Regarding nonterminating systems this means that the average throughput is maximized. While this is reasonable in high performance computing, it is usually not the primary goal in embedded systems that are subject to real-time constraints. Our scheduling algorithm solves this problem in that it not only maximizes the throughput, but also minimizes the latency. It is based on the modified critical path (MCP) algorithm developed by Wu and Gajski [17], which is widely used in practice and performs very well compared to other scheduling algorithms [18]. The basic idea of the MCP algorithm is to schedule nodes with the least ALAP times first, where ties are broken dependent on the ALAP times of the nodes' descendants. The algorithm comprises two steps[2]: In the first step, a list is created for each node containing the ALAP times of the node itself and all its descendants in ascending order. In the second step, the nodes are scheduled one after the other according to the lexicographical order of the obtained lists.

Given a set of m processors, we represent a schedule by two functions, the processor allocation function $P : V \to \{1 \dots m\}$ and the start time function $S : V \to \mathbb{N}$. These two functions specify for each node on which processor and at what time it shall be executed. As usual in static scheduling, we neglect communication costs if two adjacent nodes are scheduled on the same processor. Thus, the communication costs $c_p(u, v)$ when scheduling node v on processor p are zero if $P(u) = p$ and $c(u, v)$ otherwise. The earliest point of time v can be executed is then defined as follows:

$$\text{earliest}_p(v) = \max\{S(u) + r(u) + c_p(u, v) \mid (u, v) \in E\}$$

However, executing v at a given point of time t is only possible if it does not overlap with any other node scheduled on processor p:

$$\text{feasible}_p(v, t) = \forall w \in V.\, (P(w) = p) \to (t + r(v) \leq S(w)) \vee (S(w) + r(w) \leq t)$$

Consequently, the start time of v is the earliest feasible time:

$$\text{start}_p(v) = \min\{t \in \mathbb{N} \mid t \geq \text{earliest}_p(v) \wedge \text{feasible}_p(v, t)\}$$

Figure 4 shows our algorithm.[3] In contrast to MCP, it takes into account the unfoldings of the original ETG, which is the key to minimizing the latency. For that purpose, the subgraphs $V_1 \cup V_2 \cup \dots \cup V_n$ are processed in ascending order (lines 3–11). This way, it is guaranteed that no nodes are scheduled before all nodes of the previous unfolding have been scheduled. For each node v inside a subgraph, the algorithm creates a list l_v (lines 5–8) containing v and its successors sorted by their ALAP times. Once the nodes of a subgraph have been processed, these lists are sorted lexicographically and appended to the global list L (line 10). Finally, the nodes are scheduled in a greedy fashion (lines 12–23) w.r.t. their start times (lines 16–21).

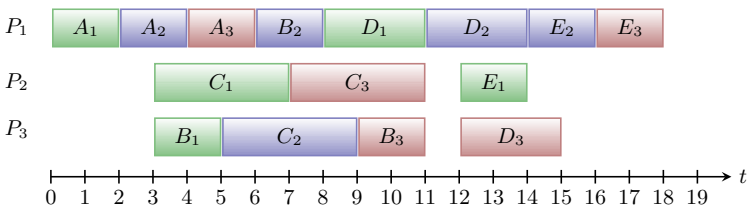
Figure 5 depicts two schedules for the ETG of Fig. 3, one computed with the classical MCP algorithm and the other one with our algorithm (LMCP). In both cases, we assumed a system with three processors. It should be emphasized that nodes belonging

[2] See *MCP Revisited* by M.-Y. Wu (`http://www.ece.unm.edu/~wu/mcp/mcp.pdf`).

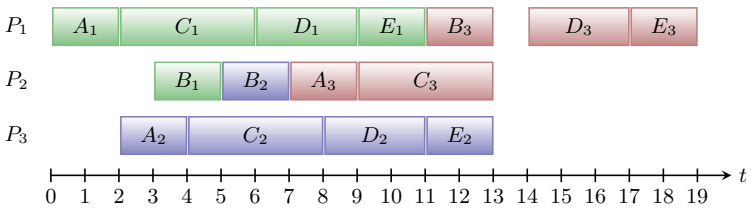
[3] An expression $[\, x \mid p(x) \,]$ denotes a list of elements each satisfying the predicate p. The second, optional parameter of the function sort is a binary predicate specifying the sorting order.

```
1  function LMCP(V_1 ∪ V_2 ∪ ··· ∪ V_n, E, r, p, c)
2    L := [];
3    // process subgraphs in ascending order (n = number of unfoldings)
4    for i := 1 to n do
5      for each v ∈ V_i do
6        // sort node and its descendants according to their ALAP times
7        l_v := sort([ w | v ⇝ w ], (a, b) → alap(a) < alap(b));
8      end for
9      // sort lists lexicographically and append them to L
10     L := append(L, sort([ l_v | v ∈ V_i ]));
11   end for
12   (P, S) := (∅, ∅);
13   while L ≠ [] do
14     v := head(head(L));
15     (P(v), S(v)) := (0, ∞);
16     // determine processor with least start time (m = number of processors)
17     for p := 1 to m do
18       if start_p(v) < S(v) then
19         (P(v), S(v)) := (p, start_p(v));
20       end if
21     end for
22     L := tail(L);
23   end while
24   return (P, S);
25 end function
```

Fig. 4. Levelized modified critical path algorithm (LMCP)



(a) Modified critical path algorithm (MCP)



(b) Levelized modified critical path algorithm (LMCP)

Fig. 5. Schedules for the unfolded ETG of Fig. 3 using three processors

to different unfoldings such as A_1 and A_2 might be executed on different processors, since they represent different tasks. Moreover, note that execution of B_1, C_1, and C_2 might overlap, since B and C are independent of each other and C is a parallel node. The maximum latency for the schedule in Fig. 5 (a) is 14 time units, whereas processing an element according to Fig. 5 (b) requires at most 12 time units.

4 Implementation

We have implemented our approach on top of a framework called FluenC [19,20], which allows developers to easily model parallel stream processing applications in C++. Typical fields of application are digital signal and image processing. As a major advantage, FluenC is based on a deterministic model of computation (dataflow process networks [21]), which simplifies testing and debugging. Determinism is particularly important in safety-critical systems, where high demands are put on correctness and reliability. For efficiency reasons, most of the classes and functions are implemented using templates. This also facilitates type safety: connecting the output of a pipeline stage to an input of a different type results in a compile-time error. Moreover, FluenC is completely lock-free, and hence does not suffer from costly synchronization operations. Figure 6 sketches the implementation of our running example.

```
...
// Serial node with input of type S and output of type T
class B: public network_t::serial<in<S>, out<T>> {
public:
  void operator()(const S& input, T& output) const {
    // computations
  }
} myB;
...
// Connect output of B with first input of D (port 0)
myB >> myD.port<0>();
```

Fig. 6. Excerpt from the implementation of the ETG shown in Fig. 2

In order to determine the runtimes of the nodes, we execute the resulting graph several times on the target processor and use the measured information for scheduling. In a second step, we create a list for each processor (core) containing the nodes to be executed. These lists are then repeatedly processed in parallel (a barrier at the end of the schedules synchronizes successive runs). Data dependencies are resolved dynamically, i.e., a node is not executed until all input data is available. For that purpose, each node has an associated table in which the incoming data is stored depending on the number of the unfolding.[4] If a processor (core) is ready, but some data is still missing, it waits until the next table entry is complete.

[4] Simply speaking, the columns represent the inputs and the rows the unfoldings.

5 Experimental Results

To evaluate our approach, we parallelized three applications, a sorting network (SORT), a fast Fourier transform (FFT) for digital signal processing, and an image recognition application (IMGR) developed at Siemens for industrial automation. The latter is used to identify moving objects in a stream of video images (cf. Sect. 1). The image processing steps range from simple local filters to complex operations, e.g. the computation of connected components and image compression. All applications exhibit a nonlinear structure and contain parallel stages.

We executed the benchmarks using self-timed as well as dynamic scheduling in order to assess the performance characteristics of both approaches. For dynamic scheduling we employed the scheduler of Intel's Threading Building Blocks[5], a state-of-the-art C++ library for parallel programming. The scheduler is based on the work stealing paradigm [6], which has proven to be very efficient in practice. To obtain comparable results, we used the same platform in both scenarios and only exchanged the underlying scheduler. All experiments were performed on a system with two quad-core Xeon E5440 processors with 2.83 GHz and 6 GB RAM running Linux.

For each benchmark, we varied the number of processor cores from one to eight and the number of unfoldings from one to ten. Moreover, we executed each benchmark 5000 times, resulting in a total number of approx. 2.4 million measured latency values. Table 1 shows the results, where L_{seq} denotes the latency (in milliseconds) and T_{seq} the throughput (elements per second) for sequential execution. For the parallel implementations, m specifies the number of processor cores and n the number of unfoldings. Besides the minimum, average, and maximum latency, we also list the standard deviation L_{dev}, which serves as a measure for the degree of timing predictability.

Table 1. Experimental results

Bench.	L_{seq}	T_{seq}	Scheduling	Objective	m	n	L_{min}	L_{avg}	L_{max}	L_{dev}	T
SORT	44	23	self-timed	min. L_{max}	2	1	33	39	67	2.8	25
				sat. T	6	9	47	94	180	17.0	38
			dynamic				57	222	421	49.6	40
				max. T	5	10	43	231	619	58.2	43
FFT	11	93	self-timed	min. L_{max}	6	1	6	6	10	0.2	167
				sat. T	8	4	10	11	26	0.6	343
			dynamic				10	15	28	1.1	271
				max. T	6	8	8	26	52	7.6	306
IMGR	84	12	self-timed	min. L_{max}	2	5	53	54	55	0.2	22
				sat. T	8	4	74	78	95	1.3	45
			dynamic				91	105	152	6.8	38
				max. T	7	10	104	192	340	34.3	52

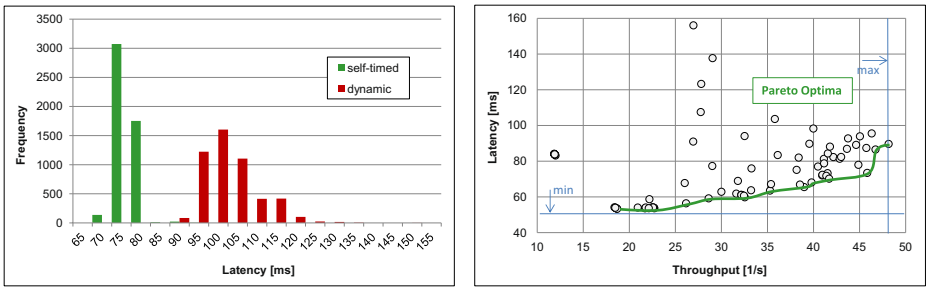
[5] `http://threadingbuildingblocks.org/`

We considered three different objectives in Table 1 regarding latency and throughput in order to summarize the results. For self-timed scheduling, we determined a combination of m and n such that the maximum latency L_{max} is minimal. Additionally, we determined when the throughput saturates, i.e., higher values for m or n do not yield significantly higher values T. To provide a direct comparison, we also list the results for dynamic scheduling with the same values for m and n. Finally, we determined a combination of m and n such that the throughput using dynamic scheduling is maximal.

SORT. Using self-timed scheduling, the average latency L_{avg} of 39 ms is slightly less than in the sequential case (89%) and the throughput of 25 is slightly higher (109%). However, it should be noted that the maximum latency is 67 ms (only 44 ms in the sequential case). Nevertheless, the standard deviation L_{dev} of 2.8 ms is comparatively low. The remaining results of SORT indicate that this benchmark does not respond well to pipelined execution, no matter which scheduling variant is employed. Even with throughput optimized dynamic scheduling, the maximum throughput is only 43 (187%) on five cores, which is relatively low compared to the other benchmarks.

FFT. If latency reduction is the primary objective, self-timed scheduling yields good results: L_{avg} is with 6 ms significantly lower than in the sequential case (55%), while still a high throughput of 167 compared to 93 (180%) is achieved. The standard deviation of 0.2 ms is remarkably low. An interesting result is the saturated throughput of 343 elements per seconds using self-timed scheduling compared to the maximum throughput achievable with dynamic scheduling of 306 elements per second. This is presumably due to the fact that the data chunks to be processed are rather small, which results in a high scheduling overhead. To sum up, self-timed scheduling pays off in significantly reduced, highly predictable latencies combined with increased throughput.

IMGR. The IMGR benchmark responds well to self-timed scheduling with respect to latency minimization: All latency values are clearly below the sequential case (55 ms in the worst case instead of 84 ms). This is merely 1 ms more than L_{avg} and 2 ms more than L_{min}. Together with the very low standard deviation 0.2 ms, these results indicate that self-timed scheduling offers highly predictable latencies for this benchmark and almost twice the throughput (183%) using two processor cores. The saturated throughput is with 45 elements per second slightly below what is achievable using dynamic scheduling (52 elements per second), whereas L_{max} = 95 ms is only a fraction of the 340 ms required to achieve such a throughput. Again, the standard deviation is very low (L_{dev} = 1.3 ms). Figure 7 (a) gives a more detailed picture of the latency distribution for both scheduling variants ($m = 8$ and $n = 4$). The measured values for self-timed scheduling span a range of 21 ms (from 74 to 95 ms), where the majority of the runs completed within 75 and 85 ms, which is a time window of only 10 ms. The latencies in dynamic scheduling are distributed over a much wider range of 61 ms (from 91 to 152 ms). Figure 7 (b) illustrates the relationship between throughput and average latency for IMGR (all combinations of m and n). As indicated by the Pareto frontier, the throughput can be increased from approx. 18 to 45 elements per second at the cost of 20 ms longer processing times (L_{avg} increases from 53 to 73 ms). The main reason for this is that our algorithm tries to fill holes in the schedules in order to reduce idle times. Consequently, increasing processor utilization may lead to fragmented schedules which in turn result in increased latencies.



(a) (b)

Fig. 7. Latency distribution for IMGR (eight cores, four unfoldings) and Pareto frontier (throughput vs. average latency) using self-timed scheduling

6 Summary and Conclusion

Besides high performance, multicore processors offer several advantages for embedded systems, in particular reduced power consumption and heat dissipation. However, the migration of sequential software to multicore processors poses serious challenges to the developers. From an industrial perspective, one of the major challenges is the protection of investment, i.e., the reuse of legacy code. Our experience has shown that stream-based applications can be parallelized relatively easy by means of pipelining, whereas the exploitation of data parallelism often requires significant refactoring.

Another challenge concerns the implementation of real-time systems. Most frameworks for parallel programming are optimized for throughput and disregard latency. In fact, the employed schedulers are usually not fair so that some tasks may even starve. To solve this problem, we proposed a polynomial-time algorithm for self-timed scheduling of nonlinear pipelines with parallel stages. It is based on the modified critical path algorithm, but takes into account the latency. For the largest of our benchmarks, an image recognition application from industrial automation, we achieved a significant reduction of the latency with only a slight drop of throughput.

It should be emphasized that our approach is not intended for hard real-time systems. Guaranteeing safe upper bounds on the execution times of parallel applications requires special hardware support and is an active area of research [22]. Another issue is the determination of an optimal number of unfoldings. Auto-tuning [23] can help to explore the design space and to find a configuration that fits the application's requirements regarding latency and throughput.

References

1. Sriram, S., Bhattacharyya, S.: Embedded Multiprocessors: Scheduling and Synchronization, 2nd edn. CRC Press (2009)
2. Lee, E., Ha, S.: Scheduling strategies for multiprocessor DSP. In: Global Telecommunications Conference, Dallas, TX, USA. IEEE (1989)
3. Davis, R., Burns, A.: A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. Technical Report YCS-2009-443, University of York, Department of Computer Science (2009)

4. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley (2007)
5. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys 31, 406–471 (1999)
6. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. In: Annual Symposium on Foundations of Computer Science (FOCS), Santa Fe, NM, USA, pp. 356–368. IEEE (1994)
7. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison Wesley (2005)
8. Ortega-Arjona, J.: Patterns for Parallel Software Design. Wiley (2010)
9. Stephens, R.: A survey of stream processing. Acta Informatica 34(7), 491–541 (1997)
10. Aldinucci, M., Torquati, M., Meneghin, M.: FastFlow: Efficient parallel streaming applications on multi-core. Technical Report TR-09-12, Università di Pisa, Dipartimento di Informatica, Italy (September 2009)
11. Otto, F., Pankratius, V., Tichy, W.F.: XJava: Exploiting parallelism with object-oriented stream programming. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 875–886. Springer, Heidelberg (2009)
12. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
13. MacDonald, S., Szafron, D., Schaeffer, J.: Rethinking the pipeline as object-oriented states with transformations. In: High-Level Parallel Programming Models and Supportive Environments (HIPS), Santa Fe, NM, USA, pp. 12–21. IEEE Computer Society (2004)
14. Bokhari, S.: Partitioning problems in parallel, pipelined, and distributed computing. IEEE Transactions on Computers 37(1), 48–57 (1988)
15. Hoang, P., Rabaey, J.: Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Transactions on Signal Processing 31(6), 2225–2235 (1993)
16. Banerjee, S., Hamada, T., Chau, P., Fellman, R.: Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. IEEE Transactions on Signal Processing 43(6), 1468–1484 (1995)
17. Wu, M.Y., Gajski, D.: Hypertool: A programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems 1, 330–343 (1990)
18. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing 59(3), 381–422 (1999)
19. Schuele, T.: A coordination language for programming embedded multi-core systems. In: International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Hiroshima, Japan. IEEE (2009)
20. Schuele, T.: Efficient parallel execution of streaming applications on multi-core processors. In: International Conference on Parallel, Distributed and Network-Based Processing (PDP), Ayia Napa, Cyprus. IEEE (2011)
21. Lee, E., Parks, T.: Dataflow process networks. Proceedings of the IEEE 83(5), 773–801 (1995)
22. Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Cassé, H., Rochange, C., Quinones, E., Uhrig, S., Gerdes, M., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., Mische, J., Paolieri, M., Wolf, J.: MERASA: Multi-core execution of hard real-time applications supporting analysability. IEEE Micro 30(5), 66–75 (2010)
23. Karcher, T., Schaefer, C., Pankratius, V.: Auto-tuning support for manycore applications: perspectives for operating systems and compilers. ACM SIGOPS Operating Systems Review Archive 43(2), 96–97 (2009)

MVA-Based Probabilistic Model of Shared Memory with a Round Robin Arbiter for Predicting Performance with Heterogeneous Workload

Ryo Kawahara, Kouichi Ono, and Takeo Nakada

IBM Research - Tokyo,
TY-S71, NBF Toyosu Canal Front, 5-6-52,
Toyosu, Koto Ward, Tokyo, 135-8511, Japan
ryokawa@jp.ibm.com

Abstract. Memory access contention can be a cause of performance problems and should be assessed at early stages of development. We devised a probabilistic model of shared memory for performance estimation. The calculation time is polynomial in the number of processors. The model is applicable for the region of high and heterogeneous bandwidth utilization. A round-robin arbiter is modeled using Mean Value Analysis (MVA) based approximations and incorporating non-linear dependence to the bandwidth utilization. To evaluate our model, estimated execution time is compared with the measured execution time of benchmark programs with memory access contention. We find a maximum error of 4.2% for the round-robin arbitration when we compensate for the burstiness of accesses.

Keywords: embedded system, shared memory, contention, simulation, analytic model, probabilistic model, UML, architecture design.

1 Introduction

Since embedded systems are increasingly large and complex, it is difficult to choose appropriate system architectures that satisfy the performance requirements. An embedded system usually has a heterogeneous architecture, which has many implementation choices, spanning hardware and software. To compete in today's markets, rapid development calls for assessing the system performance at early stages in the development process. A lightweight evaluation method is needed to prune large design spaces. Multi-processors or application-specific integrated circuits (ASICs) can be used to exploit parallelism to improve performance, but memory access contention can cause performance problems. Thus the effects of memory access contention must be considered when estimating the performance in the system architecture design phase.

There are various approaches to performance evaluation with various tradeoffs between the evaluation speed, the accuracy, and the abstraction level of

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 13–24, 2013.

each model [1]. For example, the Queuing Network (QN) model is a widely used abstract model to evaluate resource contentions. However, this model cannot handle some behaviors intrinsic to software, such as synchronization. An ESL (Electronic System Level) simulation such as SystemC [2] can also be used to evaluate the memory access contention by describing all of the access timing, but the development of such fine-grained models is expensive.

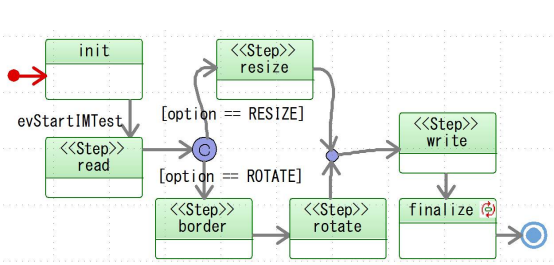
One promising approach is a hierarchical modeling [3]. The behavior at the software or application level is described by appropriate models, such as task graphs, to calculate workloads for resources, while the resource contention at the lower level is calculated using other models (such as QN model). A similar approach [4] can be used with a Unified Modeling Language (UML) model, which is easier for software designers. In addition, the parameters necessary for calculating the workloads are obtained from abstracted execution traces [5] measured on an existing reference system. This means that the parameters can be obtained at low cost, since it is often the case that the system to be developed is based on the reference system.

In the hierarchical modeling approach, the designer needs to provide a model for estimating the degree of resource contention for each shared resource such as CPUs or main memory. This paper proposes a model of shared memory for the same purpose. Since the model is used as an engine to calculate the degree of resource contention in various situations with a software model, the resource model should be versatile and lightweight. In Section 2, various methods are reviewed from this perspective. The method is described in detail in Section 3. We give experimental results in Section 4 and Section 5 concludes the study.

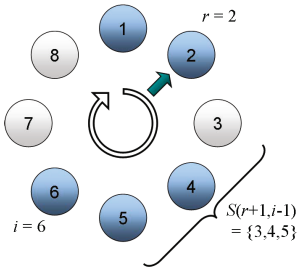
2 Related Work

Analytic methods can be used to implement lightweight performance evaluation because such methods do not rely on the detailed memory access timing information. Some approaches use probabilistic models, while others use regression models. Early research focused on the memory access contention in high performance computing (HPC) or server systems. Typical results described equivalent workloads on different processors with multiple memory modules. However, here are some key characteristics of modern embedded systems: (i) multicore processors with shared main memory, and (ii) a heterogeneous architecture with special purpose hardware or asymmetric parallel processing (AMP). This implies a method for the performance evaluation of the memory access contention should take into account these features: (i) applicability to the region of high bandwidth utilization, and (ii) different bandwidth utilization demands by different processors. Here we review some studies of the memory access contention.

Hoogendoorn [6] and Mudge [7] studied memory access contention using probabilistic models for an equal-priority arbiter. They included the effects of resubmission of rejected accesses in their analysis to increase the accuracy. An application to hierarchical modeling was demonstrated by Kawahara et al. [8]. In their method, the model counts all of the access patterns generated by N



(a) Example of UML model (State chart) for performance simulation. State with ⟨⟨Step⟩⟩ contain workload parameters.



(b) Definition of processor indexes and order of choice in a round-robin arbiter.

Fig. 1.

processors, which requires $O(2^N)$ calculations. Thus, their results are effective only for cases where all of the processors have the same access probability.

Smilauer [9] proposed a method based on Mean Value Analysis (MVA) for which the complexity of the calculations is polynomial in N. However, the results do not agree well with the simulations as the access probability increases. Sorin et al. also applied MVA to a queuing model of shared memory system with cache memories and bursty accesses [10] with good accuracy. However, their model requires at least 18 parameters for each processor and another parameter that characterizing the burstiness. Collecting these parameters from detailed simulations is time-consuming. In addition, these methods only handle equal-priority arbiters.

Another approach to model the contention involves statistics. Bobrek [11] et al. proposed a method that exploits a non-linear regression model of the resource contentions. This kind of method can be applied to various kinds of devices such as a transactional memory [12] and also has an advantage in its simulation speed. However, such methods have again high learning costs at modeling time.

Here are contributions of our work: (i) We propose a probabilistic method in which only two workload parameters are required for each processor, (ii) The contention for shared memory with a round-robin arbiter can be calculated in a polynomial order of N, (iii) The method is applicable to regions of high bandwidth utilization (total utilization ~ 0.8). These features make it possible to model the heterogeneous workloads in embedded systems.

3 Method

3.1 Hierarchical Modeling

Our probabilistic model is for a coarse-grained event-driven simulation. Executable UMLs are included in this category (Fig. 1(a)). Thus the combination of the coarse-grained event driven simulation and our model for memory access contention forms a hierarchical model. We use the word “coarse-grained” to indicate that the system behavior is described with units that correspond to large

blocks of code such as functions or tasks, and does not include instruction-level or code-level descriptions [8][5].

We call the basic unit a "step". Each step i has a workload parameter (T_i, U_i), where T_i is the step processing time and U_i is the bandwidth utilization requested when there is no memory access contention. The bandwidth utilization is defined as the ratio of the memory access throughput to its maximum determined by the hardware capacity. Assuming that a memory controller accepts either a read or a write access in one time, then the utilization can be calculated from the relation

$$U_i = \frac{M_{\mathrm{R},i}/T_i}{W_{\mathrm{R}}} + \frac{M_{\mathrm{W},i}/T_i}{W_{\mathrm{W}}}, \tag{1}$$

where W_{R} and W_{W} are the maximum throughputs of read and write memory accesses respectively, and $M_{\mathrm{R},i}$ and $M_{\mathrm{W},i}$ are the numbers of read memory accesses and of write memory accesses respectively. The numerators correspond to the actual throughputs. Note that $M_{\mathrm{R},i}$, $M_{\mathrm{W},i}$ and T_i are usually measured from an existing system (the reference system) without contention.

The step processing time on the target system T_i' is determined by adjusting T_i taking into account the resource contention. The total execution time is the sum of the T_i's along the critical execution path. This paper focuses on describing a concrete calculation procedure for the step processing time T_i'.

3.2 Overview of Probabilistic Model

In this method, we evaluate the memory access contention from the performance parameters without incorporating the information about the memory access timing. We use three approximations that are similar to those often made in theoretical analyses [7] of the memory accesses.

The first approximation is that the memory accesses occur uniformly and randomly over time, unless contention arises. This is not true in general, since the memory access pattern from a processor is normally bursty [13]. This point is discussed in Section 3.6.

The second approximation is that, within a simulation step, the memory access timings between different processors are independent of each other. Again, this is a rough approximation, because there may be periodic accesses or synchronizations. However, this approximation is necessary to reduce the order of complexity of the calculations, as explained in Sections 3.3 and 3.4.

The third approximation is that the memory access time L, which is the time for a memory controller to process a transaction without any contention, is long (e.g., $L \sim 20$ cycles). This is usually a good approximation because of the long DRAM (dynamic random access memory) latency compared to the cycle time.

The starting point of our probabilistic model is the following formula, which is similar to Amdahl's law [14]. For processor i,

$$T_i' = T_i((1 - U_i) + a_i U_i), \ (U_i' = U_i T_i / T_i'), \tag{2}$$

where $a_i = L'/L$ is the average memory access delay ratio and L' and L are the average memory access times with and without contention, respectively.

From the definition of the bandwidth utilization demand U_i in Eq. (1), this quantity corresponds to the proportion of time in which a memory controller is processing memory accesses from processor i. The formula for the bandwidth with contention U_i' comes from the fact that the total number of memory access transactions should be the same with or without the contention ($U_i T_i = U_i' T_i'$).

We call a simultaneous memory access a "collision" in this paper. In the early studies [6][7], a_i was estimated by counting all of the collision patterns. Let c_j be the state of processor j and $c = (c_1, \cdots, c_N)$. In the simple example of the access time L being one cycle, then $c_j = 1$ or 0, which correspond to the states of "accessing" or "not accessing", respectively. Then

$$a_i = \underbrace{\sum_{c_1}^{\{0,1\}} \cdots \sum_{c_N}^{\{0,1\}}}_{\text{except for } c_i} A_i(c) P(c|c_i = 1), \tag{3}$$

where $A_i(c)$ is the ratio of the access time with the collision pattern c to the access time without contention, and $P(c|c_i = 1)$ is the conditional probability of the collision pattern being c when processor i is accessing memory. The delay ratio $A_i(c)$ may include the length of a queue of memory access transactions, which depends on the arbitration scheme. We explicitly calculate a_i in the case of the round-robin arbiter with some approximations to reduce the amount of computations.

3.3 Effects of Collision with Accesses Waiting for Arbitration

Next we evaluate the probability for the states of the other processors $P(c|c_i = 1)$. Since the access patterns from different processors are assumed to be independent, $P(c|c_i = 1)$ can be factorized.

$$P(c|c_i) = \prod_{j \neq i}^{N} p_j(c_j). \tag{4}$$

For the simple case of the access time $L = 1$ cycle, we can estimate the probability of processor j being in a state c_j as $p_j(c_j = 1) = U_j$ and $p_j(c_j = 0) = 1 - U_j$ because of the approximations in Section 3.2. However, since the accesses not chosen by the arbiter will be resubmitted, there should be an increase in the probability.

To compensate for this effect, we use an idea from Hoogendoorn [6]. The probability for processor i is estimated as

$$p_j(c_j) = \begin{cases} 1 - U_j^*, & (c_j = 0), \\ U_j^*, & (c_j = 1), \end{cases} \quad \text{where } U^* = \{(T' - T) + TU\}/T'. \tag{5}$$

The modified utilization U^* includes both the original accesses TU and the increase in demand $(T' - T)$ as a result of re-submission. Note that this expression already includes the results of the calculation T', which requires an iterative

method. First we substitute $U_j^* := U_j$ into Eq. (5) as an initial value, calculate Eqns. (3), (2) and U^* in (5), and then update p_j with Eq. (5). These calculations are iterated until the values converge.

3.4 Reduction of Complexity

The sum over c in Eq. (3) requires $O(2^{N-1})$ calculations, which means it will be impossible to count all of the possible collision patterns when the number of processors N is large. The extension to the case of memory accesses taking multiple cycles is straightforward, but requires further calculations (e.g., $c_j = 1, 2, \cdots, L$ correspond to the $1, 2, \cdots, L$-th cycle from the start of a transaction). To reduce the complexity of the calculations, we use an approximation in which we replace quantities such as the queue length in $A_i(c)$ with mean values of the quantities, as in the MVA approach by Smilauer. Since it is difficult to calculate the average rigorously, we use phenomenological approximations for each arbiter case. Then, the averaging over c can be done analytically without doing $O(2^N)$ calculations.

3.5 Round-Robin Arbitration

The round-robin arbiter has states that correspond to the choice of a processor. This choice depends on the collision patterns of the previous cycles. Our approach is to introduce a non-linear U_i-dependence in a_i. Assume that each processor has an integer index $i = 1, 2, \cdots, N$ and the round-robin arbitration is done in this order (Fig. 1(b)). Let r be the index of a processor currently chosen by the arbiter. The average delay ratio is estimated as

$$a_i = \sum_{r \neq i}^{N} \left[U_r^* \beta_i + \sum_{k}^{S(r+1,i-1)} U_k^* \right] P_{\mathrm{RR}}(r|i) + 1, \tag{6}$$

where $P_{\mathrm{RR}}(r|i)$ is the conditional probability that the arbiter chooses processor r when processor i is accessing, $S(r+1, i-1)$ is the set of processor indexes between $r+1$ and $i-1$, and β_i is the ratio of the overlap between the accesses from processor r and processor i. Terms $P_{\mathrm{RR}}(r|i)$ and β_i are explained in the following paragraphs. Intuitively speaking, the terms in $[\cdots]$ correspond to a queue length in queuing theory, but in this model, the length depends on the selection of a processor by the arbiter. Here we use U^* because the average is evaluated according to the probability in Eq. (5).

The overlap $\beta_i \in [0, 1]$ is a function of U_i. The value of $\beta_i = 1$ corresponds to the case where the accesses from processors i and r are perfectly simultaneous. Here is how we model the overlap. Let l_{queue} be the queue length of memory accesses and f_i^{st} be the probability that the interval between the memory accesses from processor i is larger than l_{queue} (See Appendix A for the explicit forms of of l_{queue} and f_i^{st}). The behavior of the overlap differs between two cases: (i) If the interval is larger than l_{queue}, then the access from processor i will not collide with the queue but may collide with a new access from processor r.

Since the timings between these accesses are independent, the average point when the access arrives will be the center (i.e., $\beta_i \to 1/2$) of the current access from processor r. (ii) If the interval is smaller than l_{queue}, then an access from processor i will collide with an access in the queue. In this case the overlap becomes larger[1] (i.e., $\beta_i \to 1$) because other accesses must be processed after the previous access from processor i. We combine these two extremes.

$$\beta_i(U_i) = f_i^{\text{st}}(U_i) \times (1/2) + (1 - f_i^{\text{st}}(U_i)) \times 1. \tag{7}$$

The probability of the round-robin arbiter P_{RR} is estimated in a similar manner. It actually depends on the state of the arbiter and the collision pattern of the previous cycle. Thus we need further approximations to estimate the probability. Our estimate is

$$P_{\text{RR}}(r|i) = f_i^{\text{st}}(U_i) \times (U_r/C) + (1 - f_i^{\text{st}}(U_i)) \times \delta_{r,o(i+1)}, \quad \left(C = \sum_r^N U_r\right). \tag{8}$$

(i) In the limit of high utilization (when $U_i \to 1$), we know that the arbiter selected processor i in a previous cycle. Then the arbiter tends to select the processor next to i in the next cycle, which can be written as $P_{\text{RR}}(r|i) \to \delta_{r,o(i+1)}$ where $\delta_{i,j}$ is Kronecker's delta and $o(a) = ((a-1) \bmod N) + 1$ is a function to normalize an index to a range $[1, N]$. (ii) In contrast, in the opposite limit of $U_i \to 0$, we know nothing about the arbiter state. We can therefore estimate the probability by its steady state[2] $P_{\text{RR}}(r|i) \to U_r/C$.

3.6 Bursty Memory Access

As discussed in Section 3.2, memory accesses may be bursty. In a bursty access pattern, there are regions of high utilization and low utilization mixed over a duration. We model this behavior by first assuming that the distribution of the high and the low utilization regions is random. Thus the average collision probability is the same as for random and uniform memory access. Second, we assume that a memory access colliding with other access belongs to the high utilization region. From these assumptions, we set $U_i \to 1$ for the overlapping β_i and the round-robin probability P_{RR} appear in Eqs. (7) and (8). We set

$$\beta_i(U_i \to 1), \text{ and } P_{\text{RR}}(r|i, U_i \to 1), \tag{9}$$

while we retain U_i as in Eq. (2).

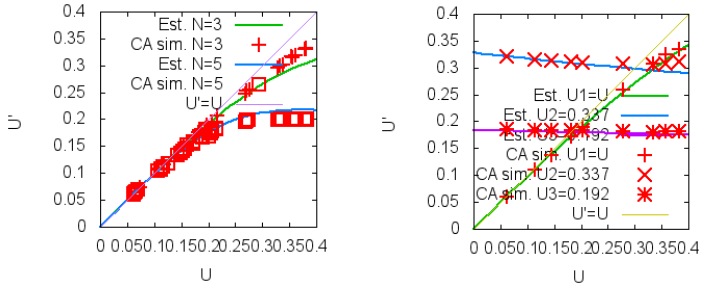
4 Evaluation

4.1 Comparison with Cycle-Accurate Simulation

Our analytic model was compared with the results of a cycle accurate simulation. The simulator we used is a commercial product (the Cadence INCISIVE

[1] This is because the interval distribution is exponential (see Appendix A) for random access patterns. The overlap is always larger than 1/2 and the above limit is true when $U_i \to 1$.

[2] This corresponds to the steady state of a Markov model whose state corresponds to a choice of a processor.



(a) $N = 3, 5$ cases. $U_i = U$ for all i.

(b) $N = 3$ case. $U_1 = U$, $U_2 = 0.337$, $U_3 = 0.192$

Fig. 2. Bandwidth utilization assigned to bus masters U' vs utilization demand U. The results of cycle-accurate simulation (CA sim.) and estimates using our probabilistic model (Est.) are shown.

simulator). We used a model that has multiple bus masters, bus slaves and a shared bus with the round robin arbitration. The burst length was set to 8 cycles. During the access, the bus is occupied. A bus master needs $8 + 12 = 20$ cycles to complete a transaction, which implies that the highest utilization generated by the bus master is $(1/20)/(1/8) = 8/20 = 0.4$. In this section, we regard a bus master as a processor and the bus is a shared resource that causes contention.

Figure 2(a) shows the cases of $N = 3$ and $N = 5$ processors. All of the bus masters generate transactions with the same probabilities. Figure 2(b) shows the case of $N = 3$ with a different utilization demand assigned to each bus master: $U_2 = 0.337$ and $U_3 = 0.192$, while $0 < U_1 < 0.4$. In both cases, the estimates by the probabilistic model agree well with the results of cycle-accurate simulations.

4.2 Comparison with Benchmark Programs

We show our experimental results in this section. The experimental system consists of a microprocessor (MPU), a DMA controller, and a shared memory. We used the Consumer program set from the MiBench benchmark suite [15] as the workload for the microprocessor, and periodic memory access patterns were generated by the DMA controller. The system was implemented on a Xilinx ML510 FPGA board [16]. Our intention for this combination of workloads was to reproduce the memory accesses in a system of software and ASICs working together.

The Consumer set in MiBench is designed to represent consumer products such as video games and digital cameras. It consists of eight programs: `jpeg`, `lame`, `mad`, `tiff2bw`, `tiff2rgba`, `tiffdither`, `tiffmedian`, and `typeset`. This set was chosen because these the programs are regarded as memory-intensive.

First we obtained the workload parameters T and U of the benchmark programs in a single processor environment. Since each program in the benchmark suite has a single function, we regard one program as a simulation step and used averaged utilization for each step. The system constants and the obtained parameters are summarized in Tables 1(a) and 1(b), respectively.

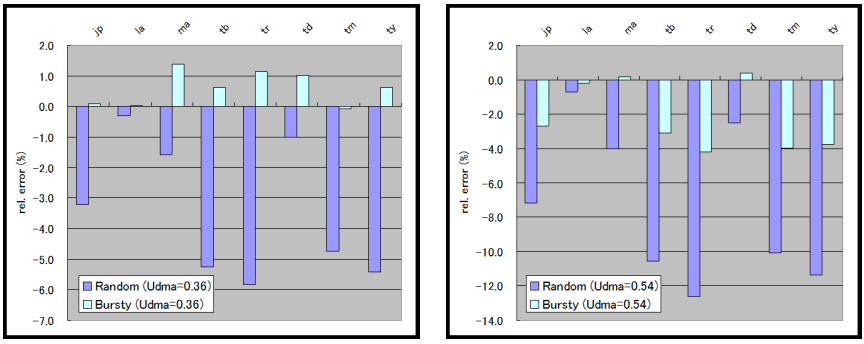
Table 1. System constants. Burst length (BL, 32-bit words), read and write throughput (R TP and W TP, in 10^6 bursts/s) are shown.

(a) System constants.

	BL	R TP	W TP
MPU	8	19.68	16.67
DMA	16	13.33	11.77

(b) Parameters

pg	T (s)	$M_{\mathrm{R}}(\times 10^6)$	$M_{\mathrm{W}}(\times 10^6)$	U
jp	0.412	0.8226	0.3258	0.153
la	62.337	16.2544	5.0265	0.0181
ma	1.117	1.7621	0.9480	0.137
tb	1.921	5.5888	3.1883	0.260
tr	4.061	12.9010	8.5785	0.305
td	3.061	2.9707	2.1318	0.0967
tm	5.204	12.8343	6.5853	0.210
ty	4.169	12.0594	7.3973	0.267

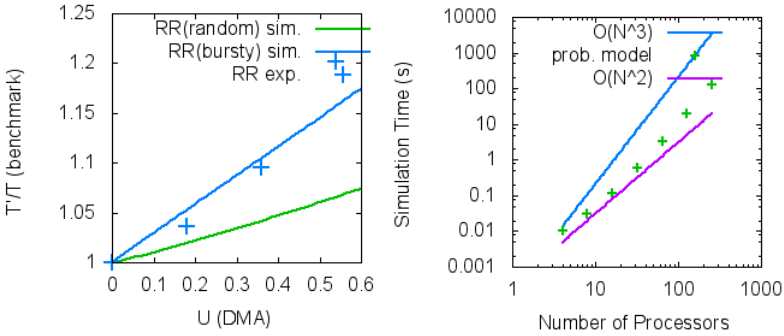


(a) $U_{\mathrm{DMA}} = 0.36$ case. (b) $U_{\mathrm{DMA}} = 0.54$ case.

Fig. 3. Errors of the model estimates in execution times from the benchmark experiment. Results of the probabilistic model for the random accesses and for the bursty accesses are shown.

When measuring the execution time, we used the `time` command. A constant time $t = 0.161$(s) (the time to execute the `time` command itself without invoking any benchmark) was subtracted from the measured time to reduce the effects of the execution overhead caused by the `time` command. The execution times in the tables are averages over 10 samples. The maximum standard deviation $\Delta T = \pm 0.019$ (s).

To measure the number of memory accesses from the microprocessor, we used a L1 cache-miss counter in an in-circuit cache emulator since any memory accesses to the shared memory are cache misses. The L2 caches were not used. The throughput constants were obtained from the specifications of the memory controller of the FPGA board. The utilization of the periodic accesses by the DMA controller was $U_{\mathrm{DMA}} = 0.180, 0.359$, and 0.539. This was adjusted by controlling the number of memory accesses within a period.



(a) Execution time of benchmark `typeset`.

(b) Simulation time of 100 runs as a function of the number of processors.

Fig. 4. Estimations by the proposed probabilistic model (RR(random) prob.) and with the extended model for bursty access (bursty) are compared to the experimental results (exp.) in (a).

In Figs 3, the results are summarized and compared with the simulation results of the probabilistic models for the random and bursty accesses. We can see that the bursty access model agrees better with the experimental results in all of the regions of U_{DMA}, and the maximum error is reduced to 4.2%, which is recorded in the case of `tiff2rgba` (tr). Figure 4(a) shows an example of the execution times of the benchmark program.

4.3 Simulation Time

In Fig. 4(b), the simulation time is plotted against the number of processors N. All of the processors have the same U, and for each plot, the simulation time is the sum of 100 runs for different Us. We used a computer with an Intel Xeon X5690 (3.46 GHz) processor. The simulation time gradually approaches $O(N^3)$.

5 Conclusion

We devised probabilistic model of shared memory with a round-robin arbiter applicable to performance estimation of embedded systems. The calculation time of the model is less than one second for $N = 100$ processors and increases as up to $O(N^3)$. The model is applicable to a shared memory of which access time is much longer than one cycle and to regions of relatively high bandwidth utilization. These features are desirable for the performance estimation of embedded systems in an early stage of development with low modeling cost and fast simulation speed.

The estimated execution time with our model is compared with the measured execution time of benchmark programs from MiBench suite with memory access

contention. Although we used the rough approximations described above, we find a maximum error of 4.2%, if we consider the burstiness of the accesses. This is acceptable accuracy if we use this model in a first-order estimation of system performance at an early stage of development [1].

Our future work will include extensions to other types of arbitration and the application of our method to real embedded systems.

Logos and Trademarks

References

1. Gries, M.: Methods for evaluating and covering the design space during early design development. Integration, the VLSI Journal 38(2), 131–183 (2004)
2. Open SystemC Initiative (OSCI): SystemC specification (2007)
3. Jonkers, H., van Gemund, A., Reijns, G.: A probabilistic approach to parallel system performance modelling. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, vol. 2, pp. 412–421 (1995)
4. Cortellessa, V., Pierini, P., Rossi, D.: Integrating software models and platform models for performance analysis. IEEE Transactions on Software Engineering 33(6), 385–401 (2007)
5. Ono, K., Toyota, M., Kawahara, R., Sakamoto, Y., Nakada, T., Fukuoka, N.: A model-based method for evaluating embedded system performance by abstraction of execution traces. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 233–244. Springer, Heidelberg (2010)
6. Hoogendoorn, C.H.: A general model for memory interference in multiprocessors. IEEE Transactions on Computers C-26(10), 998–1005 (1977)
7. Mudge, T.N., Hayes, J.P., Buzzard, G.D., Winsor, D.C.: Analysis of multiple-bus interconnection networks. Journal of Parallel and Distributed Computing 3, 328–343 (1986)
8. Kawahara, R., Nakamura, K., Ono, K., Nakada, T., Sakamoto, Y.: Coarse-grained simulation method for performance evaluation a of shared memory system. In: Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011), pp. 413–418 (2011)
9. Smilauer, B.: General model for memory interference in multiprocessors and mean value analysis. IEEE Transactions on Computers C-34, 744–751 (1985)
10. Sorin, D., Lemon, J., Eager, D., Vernon, M.: Analytic evaluation of shared-memory architectures. IEEE Transactions on Parallel and Distributed Systems 14(2), 166–180 (2003)
11. Bobrek, A., Paul, J.M., Thomas, D.E.: Stochastic contention level simulation for single-chip heterogeneous multiprocessors. IEEE Transactions on Computers 59, 1402–1418 (2010)

12. Poe, J., Cho, C.B., Li, T.: Using analytical models to efficiently explore hardware transactional memory and multi-core co-design. In: 20th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2008, pp. 159–166 (2008)
13. Darema-Rogers, F., Pfister, G.F., So, K.: Memory access patterns of parallel scientific programs. In: Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 1987, pp. 46–58. ACM, New York (1987)
14. Hennessy, J.L., Patterson, D.A.: A Quantitative Approach. In: Computer Architecture, 4th edn., pp. 1–62. Elsevier, Morgan Kaufmann Publishers (2007)
15. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: 2001 IEEE International Workshop on Proceedings of the Workload Characterization WWC-4, pp. 3–14. IEEE Computer Society, Washington, DC (2001)
16. Xilinx Inc.: Xilinx ML510 Documentation (2011)

A Appendix: Derivation of Probability f_i^{st}

Since the access timings are assumed to be random, the distribution of the access interval x has the exponential distribution $\lambda \exp(-\lambda x)$. Thus the probability f_i^{st} becomes its integral, $f_i^{\mathrm{st}} = \int_{l_{\mathrm{queue}}}^{\infty} \lambda_i \exp(-x\lambda_i) dx$, where

$$\lambda_i = \frac{U_i}{L((1-U_i) + U_i/L)} \tag{10}$$

is the rate that processor i generates an access at a cycle which is not in the state of accessing. For l_{queue}, we approximate it with the one in the simple queuing model instead of the one calculated in Eq. (6) to reduce the computational cost.

$$l_{\mathrm{queue},i} = L \sum_{j \neq i} U_j^*. \tag{11}$$

Taking the limit as $L \to \infty$ as assumed in Section 3.2, f_i^{st} quickly converges to

$$f_i^{\mathrm{st}} = \exp(-(U_i/(1-U_i)) \sum_{j \neq i} U_j^*). \tag{12}$$

MHS^2: A Map-Reduce Heuristic-Driven Minimal Hitting Set Search Algorithm

Nuno Cardoso and Rui Abreu

Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
nunopcardoso@gmail.com, rui@computer.org

Abstract. Computing minimal hitting sets (also known as set covers) for a collection of sets is an important problem in many domains (e.g., model/reasoning-based fault diagnosis). Being an NP-Hard problem, exhaustive algorithms are usually prohibitive for real-world, often large, problems. In practice, the usage of heuristic based approaches trade-off completeness for time efficiency. An example of such heuristic approaches is STACCATO, which was proposed in the context of reasoning-based fault localization. In this paper, we propose an efficient distributed algorithm, dubbed MHS^2, that renders the sequential search algorithm STACCATO suitable to distributed, Map-Reduce environments. The results show that MHS^2 scales to larger systems (when compared to STACCATO), while entailing either marginal or small runtime overhead.

Keywords: Minimal Hitting Set, Map-Reduce, Distributed Computing.

1 Introduction

Computing minimal hitting sets for a collection of constraints is an important problem in many domains (e.g., DNA analysis [11], crew scheduling [10], model/reasoning-based fault diagnosis [1,9]). The computation of minimal hitting sets can be polynomially reduced to the set cover optimization problem [6], which is known to be NP-hard. Being an NP-hard problem, the usage of exhaustive search algorithms (e.g., [9,12]), is prohibitive for most real-world problems. However, in most situations, near optimal solutions are often acceptable and approximation algorithms are used to solve this problem in a reasonable amount of time. In the particular case of model/reasoning-based fault diagnosis (MBD), which is the context of our work, the strict minimality constraint is normally relaxed[1] and heuristics are used to drive the search in order to increase the likelihood of finding the *best* minimal candidate for a particular problem instance [1,3,5,8,13].

[1] We use the term minimal in a more liberal way due to mentioned relaxation. A candidate d is said to be minimal if no other calculated candidate is contained in d.

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 25–36, 2013.

Given the paradigm shift from single core high frequency CPUs to more cost effective multi-core low frequency CPUs and the ever increasing number of platforms for parallel and distributed computing (e.g., [4]), it becomes a necessity to engineer algorithms that use such resources in order to increase the search scope of minimal hitting set (MHS) algorithms. In this paper, we propose a Map-Reduce [4] approach, dubbed MHS^2[2], aimed at computing minimal hitting sets in a parallel or even distributed fashion in order to broaden the search scope of current approaches.

This paper makes the following contributions:

- We describe the problem in the context of MBD and present a sequential algorithm to solve it.
- We introduce an optimization to the sequential algorithm, which is able to prevent a large number of redundant calculations.
- We propose MHS^2, a novel Map-Reduce algorithm for dividing the MHS problem across multiple CPUs.
- We provide an empirical evaluation of our approach, show that MHS^2 efficiently scales with the number of processing units.

The remainder of this paper is organized as follows. In Sections 2 and 3 we describe the problem as well as the sequential algorithm. In Section 4 we present our approach. Section 5 discusses the results obtained from synthetic experiments. Finally, in Section 6 we will draw some conclusions about the paper.

2 Minimal Hitting Set Problem

A set d is a minimal hitting set of a collection of non-empty sets S if and only if

$$\forall s_i \in S : (s_i \cap d \neq \emptyset) \wedge (\nexists d' \subset d : s_i \cap d' \neq \emptyset) \tag{1}$$

i.e., for each $s_i \in S$ there is at least an element that is both member of d and s_i, and no proper subset of d is a hitting set. There may be several minimal hitting sets d_k for S, which constitutes a collection of minimal hitting sets D.

Mapping this abstract description to MBD, a failed system transaction is represented by a set $s_i \in S$ containing the components involved in such transaction whereas D is the collection of diagnosis candidates. In the remainder of this paper, we capture a set of system observations in the so called hit spectra data structure [7].

Definition 1 (Hit Spectra). *Let N denote the total number of observed transactions and M denote the cardinality of the set of all components in the system, $COMPS$. We define the hit spectra data structure as being the pair (A, e), where A is a $N \times M$* activity matrix *of the system and e the* error vector*, defined as*

$$A_{ij} = \begin{cases} 1, & \textit{if component } j \in s_i \\ 0, & \textit{otherwise} \end{cases} \qquad e_i = \begin{cases} 1, & \textit{if transaction } i \textit{ failed} \\ 0, & \textit{otherwise} \end{cases} \tag{2}$$

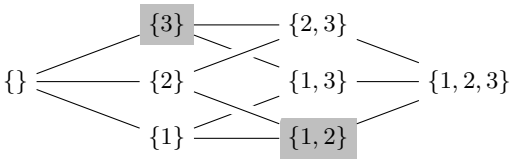
[2] MHS^2 is an acronym for Map-reduce Heuristic-driven Search for Minimal Hitting Sets.

Each row A_{i*} in the hit spectra indicates which components of $COMPS$ are members of set $s_i \in S$, whereas each column A_{*j} indicates in which sets $s_i \in S$ component $j \in COMPS$ is a member. For the particular scenario of fault localization, the goal is to find minimal hitting sets for the failing transactions. Failing transactions convey information on the possible causes of a system malfunction whereas successful transactions help the heuristic focus the search towards high potentials. The sub-collection of failing transactions, also known as conflict sets, is formally defined as $S' = \{s_i \mid e_i = 1\}$.

As an example, consider the hit spectra in Figure 1a for which all 2^M possible candidates (i.e., the power set $\mathcal{P}_{COMPS}$) are presented in Figure 1b. For this particular example, S' would be the collection $\{\{1,3\},\{2,3\}\}$ and two minimal hitting sets exist: $\{3\}$ and $\{1,2\}$. Even though the sets $\{1,3\}$, $\{2,3\}$ and $\{1,2,3\}$ are also hitting sets, they are not minimal as they can be subsumed by either $\{3\}$ or $\{1,2\}$.

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(a) Hit spectra example



(b) Hasse diagram of $\mathcal{P}_{\{1,2,3\}}$

Fig. 1. Running example

3 Staccato

A naïve, brute-force approach to compute the collection D of minimal hitting sets for S would be to iterate through the power set $\mathcal{P}$ (e.g., Figure 1b) of the set of all components in the system checking (1) whether it is a hitting set, and (2) (if it is a hitting set) whether it is minimal, i.e., not subsumed by any other set of lower cardinality (cardinality of a set d, $|d|$, is the number of elements in the set). Since many of the potential candidates turn out not to be minimal hitting sets, Staccato, the sequential algorithm which serves as foundation for our distributed algorithm, uses the Ochiai heuristic [2] in order to *focus* the search towards high-potentials, yielding significant efficiency gains. The efficiency gains are achieved by, at each stage, inspecting the component $j \in COMPS$ that, heuristically, is more likely to lead to minimal candidates. As a consequence, a large number of candidates need not to be considered as they are guaranteed to be subsumable by already discovered candidates.

In Algorithm 1 (discarding, for now, the highlighted lines) a simplified version of Staccato that captures its fundamental mechanics is presented[3]. The algorithm can perform two different tasks (lines 3–5 and 7–12), depending on whether

[3] It is important to note that the cut-off conditions were removed for the sake of simplicity and any direct implementation will not be suitable to large problems. Both the details regarding search heuristics and cut-off parameters are outside the scope of this paper. Refer to [1] for more information.

Algorithm 1. STACCATO / MHS^2 map task

Inputs:
Matrix (A, e)
Partial minimal hitting set collection D' (default: $\emptyset$)
Candidate d' (default: $\emptyset$)
Parameters:
Ranking heuristic $\mathcal{H}$
Branch level L
Load division function SKIP
Output:
Minimal hitting set collection D
Partial minimal hitting set collection D'_k

```
1  S' ← {A_i* | e_i = 1}                          # Collection of conflict sets
2  if EMPTY(S') then                              # Minimality verification task
3      if MINIMAL(D', d') then
4          D' ← PURGE_SUBSUMED(D', d')
5          D' ← D' ∪ {d'}
6  else                                           # Candidate compositions task
7      R ← RANK(H, A, e)                          # Heuristic ranking
8      for j ∈ R do
9          if ¬(SIZE(d') + 1 = L ∧ SKIP()) then   # Load Division
10             (A', e') ← STRIP(A, e, j)
11             D' ← STACCATO(A', e', D', d' + {j})
12         A ← STRIP_COMPONENT(A, j)              # Optimization
13 return D'
```

or not the candidate d' is a hitting set (d' is a hitting set if $S' = \emptyset$, entailing $\forall s_i \in S : s_i \cap d' \neq \emptyset$). In the first case, where d' is a hitting set (lines 3–5), the algorithm checks if d' is minimal (line 3) with regard to the already discovered minimal hitting set collection D'. If d' is minimal, all hitting sets in D' subsumable by d' are purged (line 4) and d' is added to D' (line 5). In the second case, where d' is not a hitting set (lines 7–12), the algorithm composes candidates in the form of $d' + \{j\}$. The order in which the components $j \in R \subset COMPS$ are selected is determined by some arbitrary heuristic $\mathcal{H}$ (line 7). This heuristic is responsible for both driving the algorithm towards high potential candidates and reducing the amount of candidates that need to be checked. Such tasks are application dependent and, in the particular case of MBD, the Ochiai heuristic [1,2] was shown to exhibit good accuracy levels. Whenever a component j is selected, a temporary (A', e') is created where all transactions s_i such that $j \in s_i$ as well as column j are omitted (function STRIP, line 10). Finally, the algorithm makes a recursive call in order to explore (A', e') with candidate $d' + \{j\}$ (line 11).

To illustrate how STACCATO works, recall the example in Figure 1a. In the outer call to the algorithm as $S' \neq \emptyset$ (i.e., candidate d' is not a hitting set), candidates in the form of $\emptyset + \{j\}$ are composed. Consider, for instance, that the result of the heuristic over (A, e) entails the ranking $(3, 2, 1)$. After composing

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(a) After STRIP$(A, e, 3)$

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(b) After STRIP$(A, e, 2)$

	1	2	3	e
s_1	1	0	1	1
s_2	0	1	1	1
s_3	1	0	1	0

(c) After STRIP$(A, e, 1)$

Fig. 2. Evolution of (A, e)

the candidate $\{3\}$ and making the recursive call, the algorithm adds it to D' as $S' = \emptyset$ (Figure 2a) and $D' = \emptyset$, yielding $D' = \{\{3\}\}$. After composing candidate $\{2\}$, a temporary (A', e') is created (Figure 2b). Following the same logic, the hitting sets $\{2, 1\}$ and $\{2, 3\}$ can be found but only $\{2, 1\}$ is minimal as $\{2, 3\}$ can be subsumed by $\{3\}$[4]. Finally the same procedure is repeated for component 1 (Figure 2c), however no new minimal hitting set is found. The result for this example would be the collection $D = \{\{1, 2\}, \{3\}\}$.

4 MHS^2

In this section we propose MHS^2, our distributed MHS search algorithm. The proposed approach can be viewed as a Map-Reduce task [4]. The map task, presented in Algorithm 1 (now also including highlighted lines), consists of an adapted version of the sequential algorithm just outlined.

In contrast to the original algorithm, we added an optimization that prevents the calculation of the same candidates in different orders (line 12), as it would be the case of candidates $\{1, 2\}$ and $\{2, 1\}$ in the example of the previous section. Additionally, it generalizes over the optimization proposed in [1], which would be able to ignore the calculation of $\{2, 3\}$ but not the redundant reevaluation of $\{1, 2\}$. The fundamental idea behind this optimization is that after analyzing all candidates that can be generated from a particular candidate d' (i.e., the recursive call), it is guaranteed that no more minimal candidates subsumable by d' will be found[5]. A consequence of this optimization is that, as the number of components in (A, e) is different for all components $j \in R$, the time that the recursive call takes to complete may vary substantially.

To parallelize the algorithm, we added a parameter L that sets the *split-level*, i.e., the number of calls in the stack minus 1 or $|d'|$, at which the computation is divided among the processes. When a process of the distributed algorithm reaches the target level L, it uses a load division function (SKIP) in order to decide which elements of the ranking to skip or to analyze. The value of L implicitly controls the granularity of decision of the load division function at the

[4] Actually, STACCATO would not compose candidate $\{2, 3\}$ due to an optimization that is a special case of the one proposed in this paper (see Section 4).

[5] Visually, using a Hasse diagram (Figure 1b), this optimization can be represented by removing all unexplored edges touching nodes subsumable by d'. As every link represents an evaluation that would be made without any optimizations (several links to same node means that the set is evaluated multiple times), it becomes obvious the potential of such optimization.

Algorithm 2. MHS^2 reduce task

Inputs:
Partial minimal hitting set collections $D'_1, ..., D'_K$
Output:
Minimal hitting set collection D

```
1 D ← ∅
2 D' ← SORT(⋃_{k=1}^{K} D'_k)                    # Hitting sets sorted by cardinality
3 for d ∈ D' do
4     if MINIMAL(D, d) then
5         D ← D ∪ {d}
6 return D
```

cost of performing more redundant calculations. Implicitly, by setting a value $L > 0$, all processes redundantly calculate all candidates such that $|d'| <= L$.

With regard to the load division function SKIP, we propose two different approaches. The first, referred to as *stride*, consists in assigning elements of the ranking R to processes in a cyclical fashion. Formally, a process $p_{k\in[1..np]}$ is assigned to an element $R_l \in R$ if $(l \mod np) = (k-1)$. The second approach, referred to as *random*, uses a pseudo-random generator in order to divide the computation. This random generator is then fed into an uniform distribution generator that assures that, over time, all p_k get assigned a similar number of elements in the ranking R although in random order (specially for large values of L). This method is aimed at obtaining a more even distribution of the problem across processes than *stride*. A particularity of this approach is that the seed of the pseudo random generator must be shared across process in order to assure that no further communication is needed.

Finally, the reduce task, responsible for merging all partial minimal hitting set collections $D'_{k\in[1..np]}$ originating from the map task (Algorithm 1), is presented in Algorithm 2. The reducer works by merging all hitting sets in a list ordered by cardinality. The ordered list is then iterated, adding all minimal hitting sets to D. As the hitting sets are inserted in an increasing cardinality order, it is not necessary to look for subsumable hitting sets (PURGE_SUBSUMED in Algorithm 1) in D.

5 Results

In order to assess the performance of our algorithm we implemented it in C++ using OpenMPI as the parallelization framework. All the benchmarks were conducted in a single computer with 2× Intel Xeon CPU X5570 @ 2.93GHz (4 cores each). Additionally, we generated several (A, e) by means of a Bernoulli distribution, parameterized by r (i.e., the probability that a component is involved in a row of A equals r) for which solutions have been computed with different parameters. In order to ease the comparison of results, *all* transactions in *all*

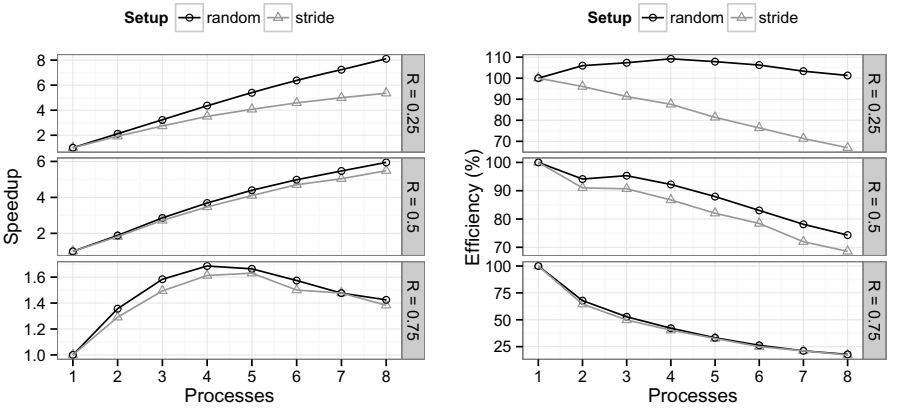


Fig. 3. Small scenarios speedup (left) and efficiency (right)

generated cases fail[6] (i.e., $S' = A$). For each set of parameters, we generated 50 inputs for each $r \in \{0.25, 0.5, 0.75\}$, and the results represent the average of the observed metrics. Both due to space constraints and the fact that the diagnosis efficiency of the algorithm has already been studied in [1], we only analyze the performance gains obtained from the parallelization.

5.1 Benchmark 1

In the first benchmark, we aimed at observing the behavior of MHS^2 for small scenarios ($N = 40, M = 40, L = 2$) where all minimal candidates can be calculated. In Figure 3, we observe both the speedup (defined as $Sup(np) = \frac{T_1}{T_{np}}$, where T_{np} is the time needed to solve the problem for np processes) and the efficiency (defined as $Ef(np) = \frac{Sup(np)}{np}$) for both the *stride* and *random* load distribution functions.

The analysis of Figure 3 shows different speedup/efficiency patterns for r values. Despite, *random* consistently outperforms *stride*. Additionally, for $r = 0.75$ and in contrast to $r \in \{0.25, 0.5\}$, the speedup/efficiency is low (note the differences in y-axis scales). The observed speedup/efficiency patterns can be explained by analyzing Figure 4 where the total runtime is divided amongst the composing tasks (calculation, communication, and the merging of results). Additionally the maximum times for calculation and communication are shown to compare them with the respective mean values. First, it is important to note that the maximum runtime for different values of r varies substantially: ≈ 200

[6] To illustrate the potential problems of having successful transactions in the test cases, consider the extreme case of a set of test cases with no failures versus a set of test cases with no nominal transactions. For the first scenario, all test cases only have one minimal hitting set (the empty set) whereas, for the second scenario, a potentially large number of minimal hitting sets may exist. As it is demonstrated in this section, the number of minimal hitting sets has a large impact in the algorithm's run-time.

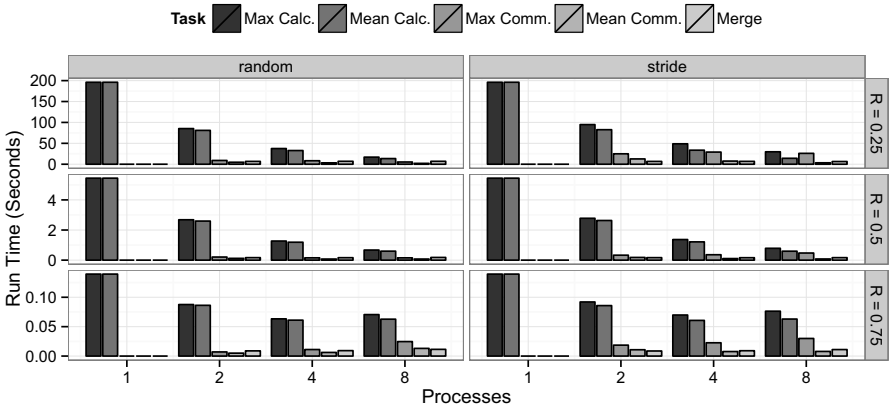


Fig. 4. Small scenarios time distribution

seconds for $r = 0.25$ vs. > 0.2 seconds for $r = 0.75$. This difference in runtimes exists due to the fact that for higher values of r, both the size and amount of minimal candidates tends to be smaller and, due to the optimization proposed in this paper, the number of candidates analyzed is also smaller. On the one hand, when the runtime is smaller, the parallelization overheads represent a higher relative impact in the performance (in extreme cases, the runtime can even increase with the number of processes). On the other hand, when both the cardinality and the amount of minimal hitting sets increase (small values of r) the parallelization overhead becomes almost insignificant. In such cases a larger efficiency difference between *stride* and *random* is observed due to a better load division by the *random* mechanism.

In Figure 4 this is visible when comparing the time bars for the maximum and mean calculation and communication times (actually, as the communication time includes waiting periods, it is dependent on the calculation time). For the *random* case, the maximum and mean calculation times are almost equal thus reducing waiting times. In contrast, in the *stride* case, the maximum and mean calculation times are uneven and, as a consequence, the communication overhead becomes higher: average ≈ 7 seconds for *random* vs. ≈ 28 seconds for *stride*. In scenarios where a large number of hitting sets exist and due to the fact of the function PURGE_SUBSUMED having a complexity greater than $O(n)$, the efficiency of the *random* load division can be greater than 100%. While good results are also observable for $r = 0.5$, the improvement is less significant.

Finally, in Figure 5 the runtime distribution of the tests is plotted. The figure shows that for the same generation parameters (r, M, N and L) the runtime exhibits a considerable variance (note that the x-axis has a logarithmic scale). It is important to note that, while the value of r has an important effect on the performance, the real key for efficiency is the problem complexity (i.e., the time needed to solve the problem). For complex enough (but still calculable) problems, the efficiency of the algorithm would increase asymptotically to 100%

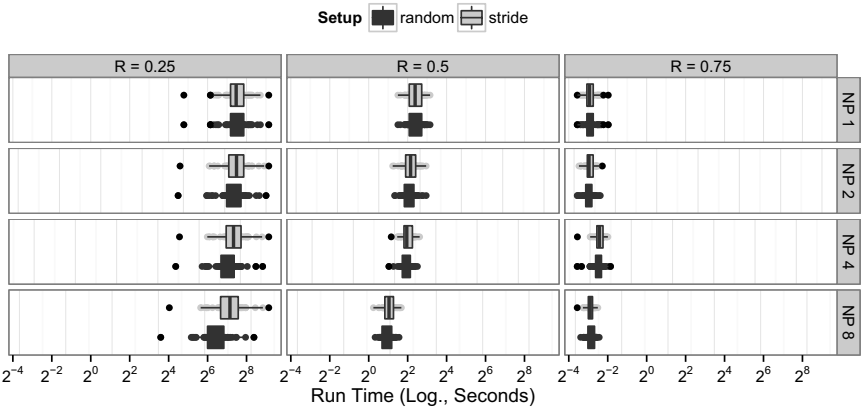


Fig. 5. Small scenarios runtime

(or even past 100%) as the polynomial parallelization overhead would eventually be overwhelmed by the exponential complexity of the problem.

5.2 Benchmark 2

The second benchmark is aimed at observing the behavior of MHS^2 for realistic scenarios ($N = 100, M = 10000, L = 2$) where it is impractical to calculate all minimal candidates. In all the following scenarios a time based cut-off was implemented and we define the metric $CSup(np) = \frac{|D_1|}{|D_{np}|}$, where $|D_{np}|$ is the number of candidates generated using np processes for the same time, henceforward referred to as *MHS speedup*. While higher values of this metric do not necessarily mean higher diagnosis capabilities[7], in most cases they are positively correlated due to the usage of the heuristic ranking function (see [1]).

Figures 6 and 7 show the results of computing candidates for big problems for runtimes $rt \in \{1, 2, 4, 8, 16\}$ and $np \in [1..8]$ (entailing a total runtime of $rt \times np$). It is clear that, for big problems, there is no significant difference in terms of the amount of generated candidates between *random* and *stride*. Figure 6 shows that for $r \in \{0.25, 0.5\}$ the MHS speedup scales well with the number of processes, however as the time increases, it becomes harder to find new minimal candidates (Figure 7). Regarding $r = 0.75$ we see that the MHS speedup pattern is not as smooth as for the former cases. Additionally it is important to note that, in contrast to the cases $r \in \{0.25, 0.5\}$, for $r = 0.75$, the number of candidates generated by $np = 8$ is smaller than for all other cases (Figure 7). This is due to the fact that for higher values of r both the cardinality and the number of minimal candidates becomes smaller, enabling the algorithm to explore a larger percentage of the search tree. As a consequence, and due to the limitations of an heuristic search, it happens that some of the candidates found first are

[7] As an example consider a set of failures for which the true explanation would be $d = \{1\}$ and two diagnostic collections $D^1 = \{\{1\}\}$ and $D^2 = \{\{1, 2\}, \{1, 3\}\}$. While $|D^2| > |D^1|$, D^1 has better diagnostic capabilities than D^2 as $d \in D^1 \wedge d \notin D^2$.

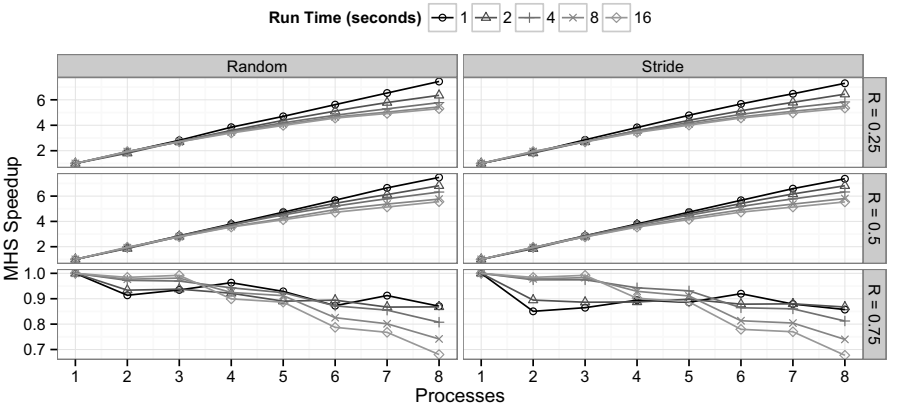


Fig. 6. Big scenarios MHS speedup

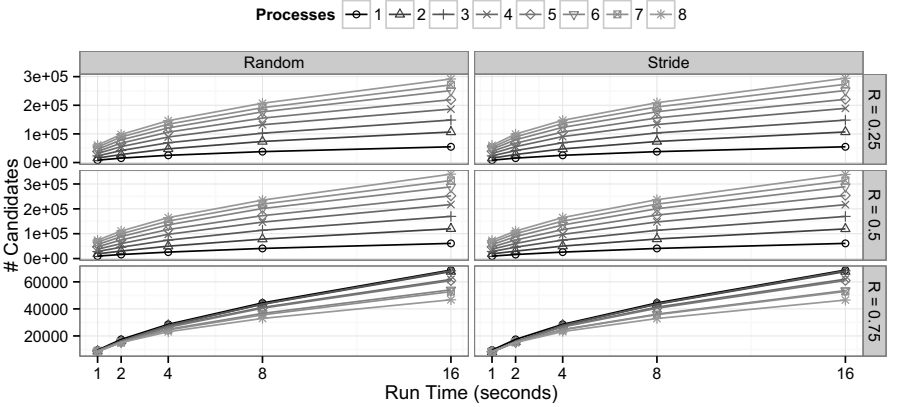


Fig. 7. Big scenarios number of candidates

subsumable by candidates found later, reducing the candidate collection size over time.

5.3 Benchmark 3

The final benchmark is aimed at observing the influence of parameter L in the number of generated candidates. In this benchmark we calculated candidates for both big and small problems using $np = 8$ and $rt = 10$ and $L \in [1..10]$. The analysis of Figure 8 reveals the great impact L has on the number of generated candidates. In the conducted experiments no MHS speedup lesser than 1 was observed, meaning that it should be a sensible choice to set L to be greater than 1. Optimal selection of values for L yielded an eightfold performance improvement for all the big scenarios. In the small scenarios, this improvement is still observable but with lesser magnitude. For the small scenarios with $r = 0.75$, L play no apparent role as the MHS speedup is always 1. This is due to the fact that all minimal candidates were always calculable within the defined time

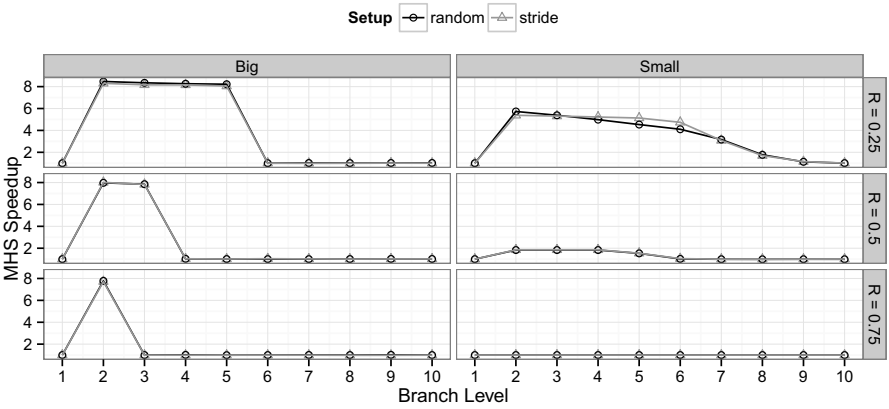


Fig. 8. Level parameter impact

frame. A closer inspection of the data revealed an isolated threefold speedup peak at $L = 2$.

Another interesting pattern is the correlation between the Bernoulli distribution parameter r and the number of near-optimal values for L. This can be explained by using the argument relating r and the candidate size. The average candidate sizes (in optimal conditions) for $r \in \{0.25, 0.5, 0.75\}$ were $\{8, 6, 3\}$ for the small scenarios and $\{6, 4, 3\}$ for the large scenarios. If we observe, for each plot, the last value of L for which the performance is near optimal we see that it matches the average candidate size minus 1. Even though several levels may be near optimal, it is better to use the smaller still near optimal value for L ($L = 2$ for all the conducted experiments) as it implies less redundant calculation with an acceptable level of granularity for load division.

As a final note, although all benchmarks were conducted within a single host, implying low communication latencies, we expect that the algorithm is able to efficiently perform in distributed environments. In the conducted experiments, the communication sizes (using a non-compressed binary stream) were bounded by a 3.77 megabytes maximum.

6 Conclusions

In this paper, we proposed a distributed algorithm to compute minimal hitting sets for a collection of constraints, dubbed MHS^2. This algorithm is not only more efficient in single CPU scenarios than the existent sequential algorithm (Staccato) but also is able to efficiently use the processing power of multiple CPUs to calculate minimal diagnosis candidates.

The results showed that, specially for large problems, the algorithm is able to scale with negligible overhead. The usage of parallel processing power enables the exploration of a larger number of potential candidates, increasing the likelihood of actually finding the "best" hitting set for a particular instance of the problem.

Future work would include the analysis of the algorithm's performance with a larger set of computation resources as also the analysis of its performance

under a wider set of conditions. Additionally, it would be intersting to study the performance in massively parallel computing Hadoop-based infrastructures.

Acknowledgements. We would like to thank Lígia Massena, André Silva and José Carlos de Campos for the useful discussions during the development of our work. This material is based upon work supported by the National Science Foundation under Grant No. CNS 1116848, and by the scholarship number SFRH/BD/79368/2011 from Fundação para a Ciência e Tecnologia (FCT).

References

1. Abreu, R., van Gemund, A.J.C.: A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In: Symposium on Abstraction, Reformulation, and Approximation, SARA 2009 (2009)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques, TAICPART 2007 (2007)
3. de Kleer, J., Williams, B.C.: Readings in model-based diagnosis (1992)
4. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Symposium on Opearting Systems Design & Implementation, OSDI 2004 (2004)
5. Feldman, A., Provan, G., Van Gemund, A.: Computing minimal diagnoses by greedy stochastic search. In: AAAI Conference on Artificial intelligence, AAAI 2008 (2008)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness (1990)
7. Harrold, M.J., Rothermel, G., Wu, R., Yi, L.: An empirical investigation of program spectra. In: Program Analysis for Software Tools and Engineering, PASTE 1998 (1998)
8. Pill, I., Quaritsch, T.: Optimizations for the boolean approach to computing minimal hitting sets. In: European Conference on Artificial Intelligence, ECAI 2012 (2012)
9. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32(1) (1987)
10. Rubin, J.: A Technique for the Solution of Massive Set Covering Problems, with Application to Airline Crew Scheduling (1973)
11. Ruchkys, D.P., Song, S.W.: A parallel approximation hitting set algorithm for gene expression analysis. In: Symposium on Computer Architecture and High Performance Computing (2002)
12. Wotawa, F.: A variant of Reiter's hitting-set algorithm. Information Processing Letters 79(1) (2001)
13. Zhao, X., Ouyang, D.: Improved algorithms for deriving all minimal conflict sets in model-based diagnosis. In: Huang, D.-S., Heutte, L., Loog, M. (eds.) ICIC 2007. LNCS, vol. 4681, pp. 157–166. Springer, Heidelberg (2007)

Handling Parallelism in a Concurrency Model

Mischael Schill, Sebastian Nanz, and Bertrand Meyer

ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Programming models for concurrency are optimized for dealing with nondeterminism, for example to handle asynchronously arriving events. To shield the developer from data race errors effectively, such models may prevent shared access to data altogether. However, this restriction also makes them unsuitable for applications that require data parallelism. We present a library-based approach for permitting parallel access to arrays while preserving the safety guarantees of the original model. When applied to SCOOP, an object-oriented concurrency model, the approach exhibits a negligible performance overhead compared to ordinary threaded implementations of two parallel benchmark programs.

1 Introduction

Writing a multithreaded program can have a variety of very different motivations [12]. Oftentimes, multithreading is a functional requirement: it enables applications to remain responsive to input, for example when using a graphical user interface. Furthermore, it is also an effective program structuring technique that makes it possible to handle nondeterministic events in a modular way; developers take advantage of this fact when designing reactive and event-based systems. In all these cases, multithreading is said to provide *concurrency*. In contrast to this, the multicore revolution has accentuated the use of multithreading for improving performance when executing programs on a multicore machine. In this case, multithreading is said to provide *parallelism*.

Programming models for multithreaded programming generally support either concurrency or parallelism. For example, the Actor model [1] or Simple Concurrent Object-Oriented Programming (SCOOP) [10,11] are typical concurrency models: they are optimized for coordination and event handling, and provide safety guarantees such as absence of data races. Models supporting parallelism on the other hand, for example OpenMP [7] or Chapel [5], put the emphasis on providing programming abstractions for efficient shared memory computations, typically without addressing safety concerns.

While a separation of concerns such as this can be very helpful, it is evident that the two worlds of concurrency and parallelism overlap to a large degree. For example, applications designed for concurrency may have computational parts the developer would like to speed up with parallelism. On the other hand, even simple data-parallel programs may suffer from concurrency issues such as data races, atomicity violations, or deadlocks. Hence, models aimed at parallelism

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 37–48, 2013.

could benefit from inheriting some of the safety guarantees commonly ensured by concurrency models.

This paper presents a library-based approach for parallel processing of shared-memory arrays within the framework of a concurrency model. To achieve this, the data structure is extended with features to obtain *slices*, i.e. contiguous data sections of the original data structure. These data parts can be safely used by parallel threads, and the race-freedom guarantee for the original data structure can be preserved.

The approach is applied to SCOOP [10,11], a concurrency model implemented on top of the object-oriented language Eiffel [8]. A performance evaluation using two benchmark programs (parallel Quicksort and matrix multiplication) shows that the approach is as fast as using threads, and naturally outperforms the original no-sharing approach. While SCOOP lends itself well to our approach, the basic idea can be helpful for providing similar extensions to Actor-based models.

The remainder of the paper is structured as follows. Section 2 describes the problem and the rationale of our approach. Section 3 presents the slicing technique. Section 4 provides the results of the performance evaluation. Section 5 describes related work and Section 6 concludes with thoughts on future work.

2 Performance Issues of Race-Free Models

To help conquer the complexity of nondeterministic multithreading, programming models for concurrency may provide safety guarantees that exclude common errors by construction. In Erlang [2] for example, a language famous for implementing the Actor model [1], there is no shared state among actors; hence the model is free from data races.

In a similar manner, the object-oriented concurrency model SCOOP [10,11] does not allow sharing of memory between its computational entities, called *processors* (an abstraction of threads, processes, physical cores etc). More specifically, every object in SCOOP belongs to exactly one processor and only this processor has access to the state of the object. A processor can however be instructed to execute a call on behalf of another processor, by adding the call to the processor's *request queue*. Also this regime offers protection from data races.

Unfortunately, the strict avoidance of shared memory has severe performance disadvantages when trying to parallelize various commonplace computational problems. As an example, Listing 1 shows an in-place Quicksort algorithm written in SCOOP. Every time the array is split, a new worker is created to sort its part of the array. The workers `s1` and `s2` and the array `data` are denoted as **separate**, i.e. they reference an object that may belong to another processor. By creating a separate object, a new processor is spawned. Calls to a separate object are only valid if the processor owning the separate object is *controlled* by the current processor, which is guaranteed if the separate object appears in the argument list, hence the `separate_sort`, `get`, and `swap` features. Each processor can only be controlled by one other processor at a time, thereby ensuring freedom from data races.

```
data: separate ARRAY[T]
lower, upper: INTEGER

make (d: separate ARRAY[T]; n: INTEGER)
    do
        if n > 0 then
            lower := d.lower
            upper := d.lower + n - 1
        else
            upper := d.upper
            lower := d.upper + n + 1
        end
        data := d
    end

sort
    local i, j: INTEGER; s1, s2: separate SORTER[T]
    do
        if upper > lower then
            pivot := get (data, upper)
            from i := lower; j := lower until i = upper loop
                if get(data, i) < pivot then
                    swap(data, i, j)
                    j := j + 1
                end
                i := i + 1
            end
            swap (data, upper, j)
            create s1.make (data, j - lower)
            create s2.make (data, j - upper)
            separate_sort(s1, s2)
        end
    end

get (d: separate ARRAY[T]; index: INTEGER): T
    do Result := data[index] end

swap (data: separate ARRAY[T]; i, j: INTEGER)
    local tmp: T do tmp := d[i]; d[i] := d[j]; d[j] := tmp end

separate_sort (s1, s2: separate SORTER[T])
    do s1.sort; s2.sort end
```

Listing 1. SORTER: In-place Quicksort in SCOOP

The execution of this example exhibits parallel slowdown: a sequential version outperforms the algorithms for most workloads. This has two main reasons:

1. Every call to the data array involves adding the call to the request queue, removing the call from the request queue, and sending back the result; this creates a large communication overhead.
2. Only one of the workers at a time can execute the `get` and `swap` features on the array because they require control of the processor handling the array; this serialization prohibits the algorithm from scaling up.

The same issues occur in a broad range of data-parallel algorithms using arrays. Efficient implementations of such algorithms are impossible in race-protective concurrency models such as SCOOP, which is unacceptable. Any viable solution

to the problem has to get rid of the communication overhead and the serialization. There are two general approaches to this problem:

1. Weaken the concurrency model to allow shared memory without race protection, or interface with a shared memory language. The programmers are responsible to take appropriate synchronization measures themselves.
2. Enable shared memory computations, but hide it in an API that preserves the race-freedom guarantees of the concurrency model.

The first approach shifts the burden to come up with solutions for parallelism to the programmer. Unfortunately, it also forfeits the purpose of race-protection mechanisms in the first place. Still, it is the most prominent approach taken. This paper presents a solution based on the second approach, in particular offering both race protection and shared memory performance.

3 Array Slicing

To allow the implementation of efficient parallel algorithms on arrays, the following two types of array manipulation have to be supported:

- *Parallel disjoint access:* Each thread has read and write access to disjoint parts of an array.
- *Parallel read:* Multiple threads have read-only access to the same array.

The array slicing technique presented in this section enables such array manipulations by defining two data structures, *slices* and *views*, representing parts of an array that can be accessed in parallel while maintaining race-freedom guarantees.

Slice. Part of an array that supports read and write access of single threads.
View. Proxy that provides read-only access to a slice while preventing modifications to it.

In the following we give a specification of the operations on slices and views.

3.1 Slices

Slices enable parallel usage patterns of arrays is where each thread works on a disjoint part of the array. The main operations on slices are defined as follows:

Slicing. Creating a slice of an array transfers some of the data of the array into the slice. If shared memory is used, the transfer can be done efficiently using aliasing of the memory and adjusting the bounds of the original array.
Merging. The reverse operation of slicing. Merging two slices requires them to be adjacent to form an undivided range of indexes. The content of the two adjacent slices is transferred to the new slice, using aliasing if the two are also adjacent in shared memory.

Table 1. API for slices

Creation procedures (constructors)	
`make(n: INTEGER)`	Create a new slice with a capacity of `n`
`slice_head(slice: SLICE; n: INTEGER)`	Slice off the first `n` entries of `slice`
`slice_tail(slice: SLICE; n: INTEGER)`	Slice off the last `n` entries of `slice`
`merge(a, b: SLICE)`	Create a new slice by merging `a` and `b`
Queries	
`item(index: INTEGER): T`	Retrieve the item at `index`
`indexes: SET[INTEGER]`	Indexes of this slice
`lower: INTEGER`	Lowest index of the index set
`upper: INTEGER`	Highest index of the index set
`count: INTEGER`	Number of indexes: `upper` − `lower` + 1
`is_modifiable: BOOLEAN`	Whether the array is currently modifiable, i.e. `readers` = 0
`readers: INTEGER`	The number of views on the slice
Commands	
`put(value: T; index: INTEGER): T`	Store `value` at `index`
Commands only accessible to slice views	
`freeze`	Notifies the slice that a view on it is created by incrementing `readers`
`melt`	Notifies the slice that a view on it is released by decrementing `readers`
Internal state	
`area: POINTER`	Direct unprotected memory access
`base: INTEGER`	The offset into memory

Based on this central idea, an API for slices can be defined as in Table 1. Note that we use the letter `T` to refer to the type of the array elements. After creating a new slice using `make`, the slice can be used like a regular array using `item` and `put` with the `indexes` ranging from `lower` to `upper`, although modifying it is only allowed if `is_modifiable` is true, which is exactly if `readers` is zero. Internally, the attribute `area` is a direct pointer into memory which can be accessed like a 0-based array. The `base` represents the base of the slice, which is usually 1 for Eiffel programs, but may differ when a merge results in a copy. The operations `freeze` and `melt` increment and decrement the `readers` attribute which influences `is_modifiable` and are used by views (see section 3.2).

Slicing. Like any other object, a reference to the slice can be passed to other processors. A processor having a reference to a slice can decide to create a new slice by slicing from the lower end (`slice_head`) or upper end (`slice_tail`). By doing this, the original slice transfers data to the new slice by altering the bounds and referencing the same memory if possible. Freedom of race conditions is ensured through the exclusive access to the disjoint parts.

Listing 2 shows an implementation of the `slice_head` creation procedure, taking advantage of shared memory. It copies the `lower` bound, the `base` and the memory reference of the slice `a_original`. It also sets the upper bound

```
slice_head (n: INTEGER; a_original: separate SLICE[T])
    require -- Precondition
        within_bounds: n > 0 and n <= a_original.count
        a_original.is_modifiable
    do
        lower := a_original.lower
        upper := a_original.lower + n - 1
        base := a_original.base
        area := a_original.area
        a_original.lower := a_original.lower + n
    ensure -- Postcondition
        a_original.count = old a_original.count - n
        a_original.lower = old a_original.lower + n
        a_original.upper = old a_original.upper
        lower = old a_original.lower
        upper = a_original.lower - 1
        count = n
        -- "forall i in indexes : item(i) = old a_original.item(i)"
    end
```

Listing 2. Slicing

according to the size `n` of the new slice and increases the lower bound of the original by `n`.

We use Eiffel for our implementation. Eiffel provides preconditions and postconditions, which we use to make sure only modifiable arrays are altered.

Merging. If a processor has two *adjacent* slices (the lower index of the one equals the upper index of the other plus one), calling `merge` creates a new combined slice. This transfers all the data from the old slices to the new one, making the old ones empty. If the two slices are located next to each other in memory, the transfer simply adjusts the bounds; otherwise, it copies the data into a new slice.

The implementation of merging (see Listing 3) sets the bounds according to the arguments. It then checks whether the two parts are actually next to each other in memory by checking whether the `area` and the `base` are the same. In this case, it copies the base and the memory reference. Otherwise, it allocates new memory and copies all the data of the two arguments. In the end, it empties the two arguments, setting their `count` to 0 by making `lower` = 1 and `upper` = 0.

Strategies for Slicing. The most common choice for disjoint index subsets are sets with contiguous indexes. Those subsets can be identified by their lower and upper index and resemble a normal array. A rarer case is to create the disjoint subsets according to another principle. This warrants a different implementation, which is possible by using inheritance. However, current cache architectures limit the usefulness of slices with a size smaller than a cache line.

3.2 Views

Views enable read-only access on arrays. The main operations on views are defined as follows:

```
merge (a_one, a_another: separate SLICE[T])
    require
        a_one.is_modifiable
        a_another.is_modifiable
        one.is_adjacent (a_another)
    do
        lower := a_one.lower.min(a_another.lower)
        upper := a_another.upper.max(a_one.upper)
        if  a_one.area = a_another.area and a_one.base = a_another.
            base then
            area := a_one.area
            base := a_one.base
        else
            base := lower
            -- "Copy data from the a_one and a_another to area"
        end
        a_another.empty; a_one.empty
    ensure
        lower = old a_one.lower.min(a_another.lower)
        upper = old a_one.upper.max(a_another.upper)
        a_one.count = 0 and a_another.count = 0
        -- "forall i in old a_one.indexes : item(i) = old a_one.item(i
            )"
        -- "forall i in old a_another.indexes : item(i) = old
            a_another.item(i)"
    end
```

Listing 3. Merging

Table 2. API for slice views

Creation procedures (constructors)	
`make(slice: SLICE)`	Create a new view on `slice`
Queries	
`original: SLICE[T]`	Slice this view references
`indexes: SET[INTEGER]`	Indexes of this view
`item(index: INTEGER): T`	Retrieve the item at index
`lower: INTEGER`	Lowest index of the index set
`upper: INTEGER`	Highest index of the index set
Commands	
`free`	Disconnects the view from the slice
Internal state	
`area: POINTER`	Direct unprotected memory access
`base: INTEGER`	Offset into memory

Viewing. Creating a view from a slice copies the bounds and the memory reference into the view. The original slice is no longer modifiable.

Releasing. The reverse operation of viewing. If no other views on the same slice exist, it is modifiable again. Also, the view is no longer usable.

The API for views is shown in Table 2. A processor is able to read the slice in parallel by creating a view using the `make` creation procedure. The original slice is then available as the `original` query. This also prevents all further modifications of the array unless the view is released with the `free` procedure. All the other features of views behave exactly like their counterparts in the slices.

```
make (a_original: separate SLICE[T])
    do
        a_original.freeze
        original := a_original
        lower := a_original.lower
        upper := a_original.upper
        base := a_original.base
        area := a_original.area
    ensure
        lower = a_original.lower
        upper = a_original.upper
        not a_original.is_modifiable
        -- "forall i in indexes : item(i) = a_original.item(i)"
    end
```

Listing 4. Viewing

Viewing. Creating a view basically copies the bounds and the memory reference into the view. By increasing the view counter (`readers`) using the `freeze` operation of the slice `a_original`, the original slice is no longer modifiable (see Listing 4). By calling `free` on a view, the view loses its reference to the memory of the slice and the original slice is notified through `melt` that there is one less `reader`.

Releasing. The `free` procedure redirects the `area` to 0 and sets `lower` to 1 and `upper` to 0. Therefore no access is possible at any index. In addition, the number of readers of the original decremented by a call to `melt`. This causes the original to be modifiable again if the number of readers falls to zero. Because of its simplicity, the code is omitted.

4 Performance Evaluation

To assess the performance of our approach, we apply it to two benchmark problems: to determine how well our approach works in a divide-and-conquer scenario, we choose a parallel in-place Quicksort algorithm; to determine the raw performance, we use parallel matrix multiplication. In both cases, the extension of SCOOP with the slicing technique is compared with implementations in Eiffel using only threads and without synchronization except a join at the end.

For the performance tests we use a server with four 8-core Intel Xeon E7-4830 processors and 256 GB of RAM. We ran every program 20 times and report the mean value of the running times in Table 3. The source code of the benchmarks is available online[1] as well as the support for slicing in SCOOP[2]. In the following we discuss both benchmarks and their results in detail.

[1] `https://bitbucket.org/mischaelschill/array-benchmarks`
[2] `https://bitbucket.org/mischaelschill/scoop-library`

Table 3. Mean running times (in seconds)

		Number of cores					
		1	2	4	8	16	32
Quicksort	Slicing	157.4	147.1	81.9	66.4	59.9	59.2
	Threads	158.6	145.1	82.8	68.0	61.5	59.8
Matrix multiplication	Slicing	184.8	95.0	51.2	24.0	14.1	7.3
	Threads	178.0	91.7	46.6	23.6	12.6	7.3

4.1 Quicksort

Listing 5 shows the constructor of the Quicksort example implemented using slices instead of regular arrays (compare Section 2). The main difference is the usage of `slice_head` and `slice_tail` instead of storing the bounds in variables. The implementation of `sort` can stay the same, although there is no need for storing the bounds and extra features for swapping and retrieving since data is no longer separate.

```
data: SLICE[T]

make (a_data: SLICE[T]; n: INTEGER)
    do
        if n > 0 then
            create data.slice_head (a_data, n)
        else
            create data.slice_tail (a_data, -n)
        end
    end
```

Listing 5. Quicksort algorithm using slices

For the performance measurement, the Quicksort benchmark sorts an array of size 10^8, which is first filled using a random number generator with a fixed seed. The benchmarked code is similar to listing 5, but also adds a limit on the number of processors used. As evident from Figure 1, the performance characteristics of the slicing technique and threading is almost identical.

4.2 Matrix Multiplication

Listing 6 shows a class facilitating parallel multiplication of matrices, using a two dimensional version of slices and views (`SLICE2` and `SLICE_VIEW2`, implemented in a very similar fashion to the one-dimensional version discussed in Section 3.1). The worker is created using `make` which slices off the first `n` rows into `product`. The `multiply` command actually fills the slice with the result of the multiplication of the left and right matrices. Afterwards, the views are decoupled using `free`. Dividing the work between multiple workers and merging the result is left to the client of the worker.

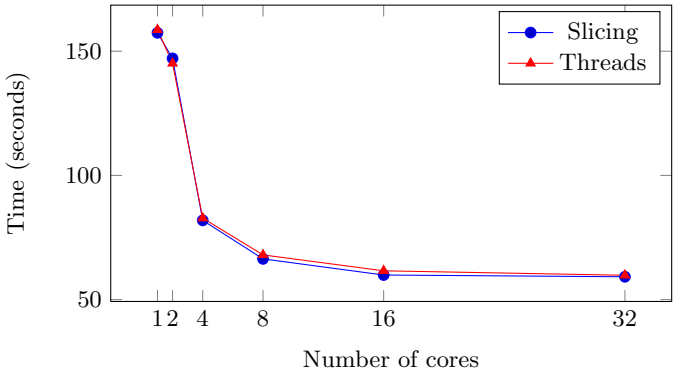


Fig. 1. Quicksort: scalability

```
left, right: SLICE_VIEW2[T]
product: SLICE2[T]

make (l, r, p: separate SLICE2[T]; n: INTEGER)
    do
        create left.make(l); create right.make(r)
        create product.slice_top (n, p)
    end

multiply
    local k, i, j: INTEGER
    do
        from i := product.first_row until i > product.last_row loop
            from j := product.first_column until j > product.
                last_column loop
                from k := left.first_column until k > left.last_column
                    loop
                    product[i, j] := product[i, j] + left[i, k] *
                        right [k, j]
                    k := k + 1
                end
                j := j + 1
            end
            i := i + 1
        end
        left.free; right.free
    end
```

Listing 6. Matrix multiplication worker using slices and views

For the performance measurement, the matrix multiplication test multiplies a 2000 to 800 matrix with an 800 to 2000 matrix. Figure 2 shows again similar performance characteristics between slicing and threads.

5 Related Work

We are not aware of any programming model supporting slicing while avoiding race conditions. However, similar means to create an alias to a subset of

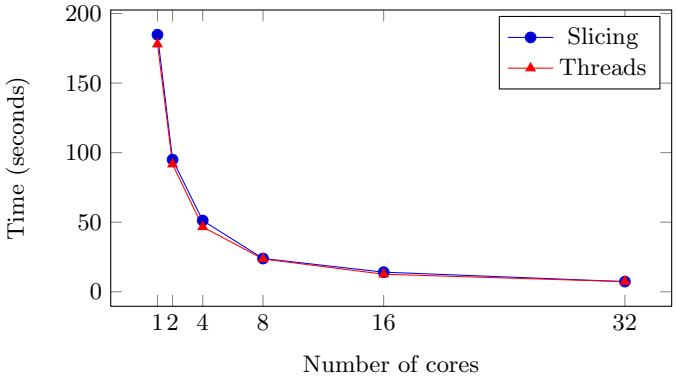


Fig. 2. Matrix multiplication: scalability

an array's content are common in most programming languages or their standard library. For example, the standard library of Eiffel as provided by Eiffel Software [8] can create subarrays. Perl [13] has language integrated support for slicing arrays. Slices and slicing are a central feature of the Go programming language [9]. However, these slicing solutions were not created with the intention of guaranteeing safe access: the portion of memory aliased by the new array/slice remains accessible through the original array, which can lead to race conditions if two threads access them at the same time.

Enabling many processors to access different parts of a single array is a cornerstone of data parallel programming models. OpenMP [7] is the de-facto standard for shared-memory multiprocessing. Its API offers various data parallel directives for handling the access to arrays, e.g. in conjunction with parallel-for loops. Threading Building Blocks [14] is a C++ library which offers a wide variety of algorithmic skeletons for parallel programming patterns with array manipulations. Chapel [5] is a parallel programming language for high-performance computation offering concise abstractions for parallel programming patterns. Data Parallel Haskell [4] implements the model of nested data parallelism (inspired by NESL [3]), extending parallel access also to user-defined data structures. In difference to our work, these approaches focus on efficient computation but not on safety guarantees for concurrent access, which is our starting point.

The concept of views is an application of readers-writers locks first introduced by Courtois, Heymans and Parnas [6], tailored to the concept of slices.

6 Conclusion

While programming models for concurrency and parallelism have different goals, they can benefit from each other: concurrency models provide safety mechanisms that can be advantageous for parallelism as well; parallelism models provide performance optimizations that can also be profitable in concurrent programming. In this paper, we have taken a step in this direction by extending a concurrency

model, SCOOP, with a technique for efficient parallel access of arrays, without compromising the original data-race freedom guarantees of the model. An important insight from this work is that safety and performance do not necessarily have to be trade-offs: results on two typical benchmark problems show that our approach has the same performance characteristics as ordinary threading.

In future work, it would be interesting to explore the relation between models for concurrency and parallelism further, with the final goal of defining a safe parallel programming approach. In particular, programming patterns such as parallel-for, parallel-reduce, or parallel-scan could be expressed in a safe manner. In order to ascertain how this API is used by programmers, empirical studies are needed.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA).

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
2. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent programming in Erlang, 2nd edn. Prentice-Hall (1996)
3. Blelloch, G.E.: NESL: A nested data-parallel language. Tech. Rep. CMU-CS-95-170, Carnegie Mellon University (1995)
4. Chakravarty, M.M.T., Leshchinskiy, R., Jones, P.S., Keller, G., Marlow, S.: Data parallel Haskell: a status report. In: Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming, pp. 10–18. ACM (2007)
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
6. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with readers and writers. Communications of the ACM 14(10), 667–668 (1971)
7. Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. IEEE Computer Science & Engineering 5(1), 46–55 (1998)
8. Eiffel Software (2013), `http://www.eiffel.com/`
9. Go programming language (2013), `http://golang.org/`
10. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)
11. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
12. Okur, S., Dig, D.: How do developers use parallel libraries? In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012, pp. 54:1–54:11. ACM (2012)
13. Perl programming language (2013), `http://www.perl.org/`
14. Reinders, J.: Intel threading building blocks – outfitting C++ for multi-core processor parallelism. O'Reilly (2007)

On the Relevance of Total-Order Broadcast Implementations in Replicated Software Transactional Memories

Tiago M. Vale, Ricardo J. Dias, and João M. Lourenço

Departamento de Informática and CITI
Universidade Nova de Lisboa, Portugal
{t.vale,ricardo.dias}@campus.fct.unl.pt,
joao.lourenco@fct.unl.pt

Abstract. Transactional Memory (TM), an attractive solution to support concurrent accesses to main-memory storage, is already being deployed by some of the major CPU and compiler manufacturers. To address scalability and dependability challenges, researchers are now combining replication, TM and certification-based protocols. To maintain consistency and ensure common transaction serialisation order, these protocols rely in a total-order broadcast primitive, usually provided by some Group Communication System (GCS). The total-order broadcast service can be implemented by different algorithms, which hold different properties. In this paper we present a detailed analysis of the impact of some algorithms implementing total-order broadcast in different TM workloads, opening up future work to improve performance of replicated TMs.

1 Introduction

The interest in research on paradigms for parallel programming increased as multi-core computers hit mainstream. Software Transactional Memory (STM) [1] as earned interest from both the academic and industrial research communities. STM relieves the programmer from the subtleties of the traditional lock-based concurrency control by adapting the familiar concept of transaction inherited from the database world. Enterprise-class STM-based applications have already been deployed in production systems[1]. These real-world applications usually hold requirements such as scalability and reliability which are commonly tackled using replication. Distributed STM has been similarly motivated as an alternative to (distributed) lock-based concurrency control in distributed systems, where the problems associated with locks are exacerbated.

Given the similarities between STM and database transactions, research on STM replication have borrowed inspiration from the literature in replicated databases. A handful of replication protocols [2,3,4,5], commonly referred to as certification-based, have been proposed and evaluated in the context of replicated

[1] https://fenix-ashes.ist.utl.pt

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 49–60, 2013.

STM systems. Replica consistency is ensured at commit time using a total-order broadcast to guarantee a common transaction serialisation order. Unfortunately, the relative overhead introduced by the certification of distributed memory transactions is much higher than the overhead introduced by the certification of distributed database transactions. On the one hand, memory transactions operate over main memory which is a very fast storage media, and on the other hand, some key features of databases require additional processing time, such as SQL parsing, plan optimisation, and secondary storage accesses.

Total-order broadcast can be implemented using different algorithms, which exhibit different properties such as latency, fairness and throughput. In this paper, and to the best of our knowledge, we present the first study of the impact that the different algorithms implementing total-order broadcast have on replicated STMs. In the remainder of this paper, we discuss certification-based protocols and their key ingredient, the total-order broadcast, in §2; followed by a discussion of its impact in replicated STM environments in §3. We proceed with a description of our implementation and discuss the experimental results in §4 and §5, respectively. We close the paper with a discussion of the related work in §6, and some concluding remarks in §7.

2 Software Transactional Memory Replication

While the transaction concept bridges the world of databases and STM, memory transactions' execution time is significantly smaller than database transactions. Memory transactions only access data in main memory, thus not incurring in the expensive secondary storage accesses characterising the latter. Furthermore, SQL parsing and plan optimisation are also absent in STM. On the other hand this increases the relative cost of remote coordination. Nonetheless, literature on replicated and distributed databases represents a natural source of inspiration when developing protocols for replicated STM. In fact, current research on replicated STM have embraced protocols commonly referred to as certification-based. These protocols rely on total-order broadcast, usually provided by some group communication system, to impose a global transaction serialisation order.

2.1 Total-Order Broadcast

Informally, a total-order broadcast ensures that messages sent to a set of recipients are delivered in the same order by all recipients. Total-order broadcast guarantees the following properties [6]: (1) *Validity*: if a correct replica TO-broadcasts a message m, then it eventually TO-delivers m; (2) *(Non-)uniform Agreement*: if a (correct) replica TO-delivers a message m, then all correct replicas eventually TO-deliver m; (3) *Uniform Integrity*: for any message m, every replica TO-delivers m at most once, and only if m was previously TO-broadcast by $\mathsf{sender}(m)$; and (4) *(Non-)uniform Total Order*: if (correct) replicas r_1 and r_2 both TO-deliver messages m and m', then r_1 TO-delivers m before m', if and only if r_2 TO-delivers m before m'. A broadcast that satisfies all these properties

except (4), *i.e.*, that provides no ordering guarantee, is instead called a reliable broadcast.

Notation. A total-order broadcast consists of two primitives, to-broadcast(m) and to-deliver(m). We say a replica r TO-broadcasts a message m when r executes to-broadcast(m), and r TO-delivers m when r executes to-deliver(m). We denote R-broadcast and R-deliver analogously for reliable broadcast. The sender of a broadcasted message m is denoted by sender(m).

There are several algorithms to implement total-order broadcast [6]. In sequencer-based algorithms one replica is elected as the sequencer and is responsible for ordering messages. For example, any replica r wanting to TO-broadcast a message m, starts by unicasting m to the sequencer, which in turn broadcasts m on behalf of r. A different approach is followed by privilege-based algorithms. These algorithms rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process.

2.2 Certification-Based Protocols

Current research in STM replication has borrowed protocols from the database literature usually classified as certification-based. In certification-based protocols, unlike classical eager replication, transactions are optimistically executed on a single replica without any remote coordination. Updates are buffered and applied at all replicas if a transaction is found to be valid and successfully commits. Replicas coordinate at commit time by way of a distributed protocol that validates (certifies) transactions and establishes a global transaction serialisation order. Thus, the outcome of a transaction's validation is the same at every replica, and the updates of valid transactions are applied at every replica in the same order. While update transactions require replica coordination at commit time, as described, read-only transactions can validate and commit locally at the host replica.

To impose a global transaction serialisation order, certification-based protocols rely on total-order broadcast to disseminate transactions at commit time. This contrasts with classical eager replication protocols based on distributed locking that potentially incur in deadlocks and suffer from communication overheads during the transaction execution phase. Depending on which transactional data is TO-broadcasted, certification protocols can be classified in two schemes: Non-Voting [7] and Voting [8]. In the Non-Voting scheme, when a transaction t executing at some replica enters the commit phase, both its write set W and read set R are TO-broadcasted. This means that each replica is able to independently validate and abort or commit t, as they are in possession of all the necessary information, *i.e.*, both W and R. Note that given the total ordering of the deliveries, all replicas will process all transactions in the same order, so the result of any transaction's validation will be the same on all replicas.

By disseminating both W and R the Non-Voting scheme requires a single communication round to commit a transaction. However, the read set is typically much larger than the write set, thus the second scheme, Voting, explores the trade-off of exchanging potentially much smaller messages (since R, typically much larger than W, is not disseminated) at the expense of requiring two communication rounds (an additional R-broadcast) instead of just one.

3 On the Relevance of Total-Order Broadcast Implementations

Consider the typical work flow of a transaction-processing thread *Th* under a certification-based protocol. *Th* executes transaction t. If t is read-only it is locally validated and committed (or aborted, thus re-executed) by *Th*. Otherwise *Th* TO-broadcasts t and waits. Upon the respective TO-delivery, *Th* validates and commits or aborts t, re-executing if aborted. We refer to the time *Th* waits, *i.e.*, the time between the TO-broadcast and the respective TO-delivery, as latency.

Intuitively, different implementations of total-order broadcast will have a different impact on latency, and thus on the performance of typical replicated STM deployments, where each replica executes roughly the same percentage of update transactions. In the sequencer-based algorithm (see §2.1) it is expected that (1) latency in replica s, assigned as sequencer, is lower than in the remaining replicas; and (2) transaction ordering is biased towards transactions from s, because unlike other replicas, s can skip an initial unicast. With privilege-based implementations, latency is expected to be similar in all replicas, as they are only allowed to broadcast during their slot. Thus, a sequencer-based total-order broadcast implementation is likely to allow a replicated STM to achieve higher performance than a privilege-based implementation at the expense of an unequal contribution of each replica to the global throughput of the system. The sequencer will likely execute much more transactions, and faster, than the other replicas. This contrasts with the privilege-based solution in which each replica is expected to contribute evenly.

4 Implementation

To implement and evaluate the relevance of different total-order broadcast implementations in replicated STMs, we have extended an existing Java STM framework [9] to support replication in a transparent way for the programmer. In this paper we will only cover the architecture and API for implementing replicated STM algorithms, whose understanding is necessary for our study. The STM layer implements an STM algorithm and exposes an API to the application with the following primitives:

```
1.  stm_commit(t):
2.    if t is not read-only and stm_validate(t)
3.        certify(t)
4.    else
5.        if not stm_validate(t)
6.            stm_abort(t)
7.  certify(t):
8.    to-broadcast(t)
9.    wait until to-deliver(t)
10.   if stm_validate(t)
11.       stm_apply(t)
12.   else
13.       stm_abort(t)
```

Fig. 1. Non-Voting Certification implementation on our framework

stm_begin(t)	Begin transaction t.
stm_commit(t)	Request commit of transaction t.
stm_abort(t)	Aborts transaction t.
stm_read(t, m)	Transaction t reads value of memory location m.
stm_write(t, m, v)	Transaction t writes value v to memory location m.

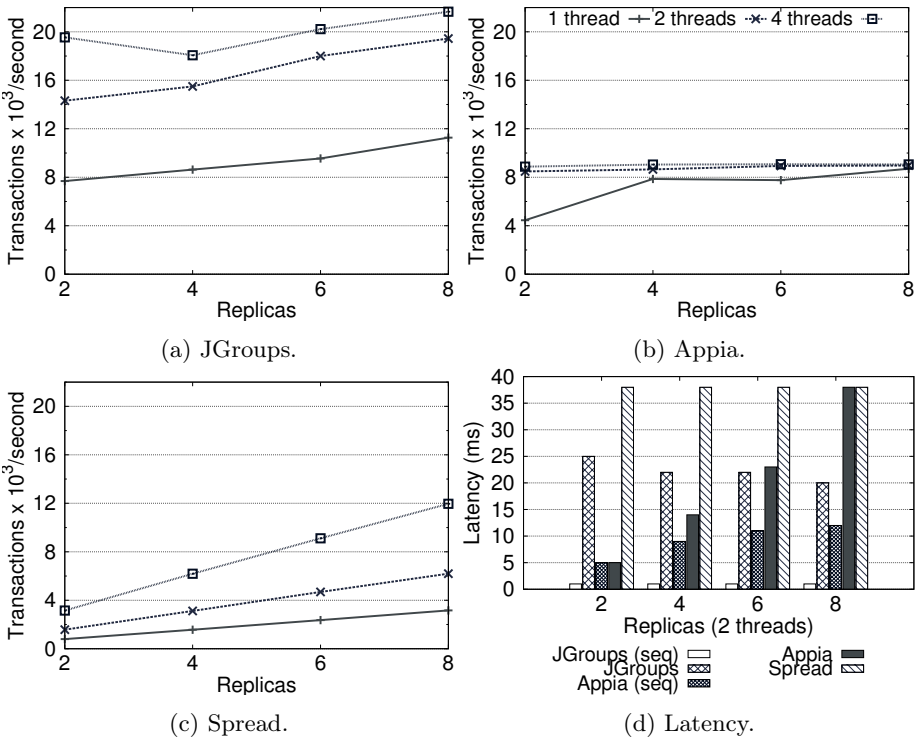
Application code defines transactions by tagging methods with an `@Atomic` annotation. These methods are bytecode-instrumented to inject calls to the underlying STM layer using the described API where appropriate. To support the distributed certification of transactions, the STM layer interacts with the certification-based protocol layer through the following API:

stm_validate(t)	Transaction t is validated against the local STM state.
stm_apply(t)	Transaction t's updates are applied to the local STM state.
certify(t)	Issues the distributed certification of transaction t.

The certify primitive allows the STM layer to trigger the distributed certification of an update transaction when stm_commit is issued by the application. When certifying a transaction t, stm_validate and stm_apply concede the certification-based protocol the ability to validate t in the local replica and apply its updates, respectively. On the bottom of the architectural stack we have a group communication system that provides the total-order broadcast primitives, to-broadcast(t) and to-deliver(t). These are used by the certification-based protocol layer to broadcast transactions for certification while ensuring a common order across all replicas. Figure 1 illustrates an implementation of the Non-Voting Certification under our framework.

5 Experimental Results

All the experiments were performed in a cluster of 8 nodes interconnected via gigabit ethernet, each one equipped with a Quad-Core AMD Opteron 2376 at 2.3 Ghz, 4×512KB cache L2, and 8 GB of RAM. The installed Java Platform was OpenJDK 6. The local STM layer of our infrastructure used the TL2 algorithm [10]. The certification-based protocol was the Non-Voting scheme. With regard to the underlying group communication system providing the total-order broadcast primitives needed by the certification-based protocol, three different implementations were considered: JGroups, Appia [11] and Spread. Switching between each group communication system (GCS) is done by parametrisation when executing the target program, hence no code rewriting whatsoever is needed.



(a) JGroups. (b) Appia.

(c) Spread. (d) Latency.

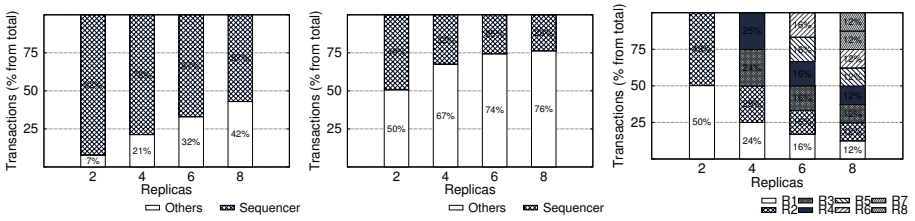
Fig. 2. Throughput and total-order broadcast latency in Red-Black Tree, configured with initial size 32078, range 131072 and 10% updates

JGroups is a well-known toolkit used in several projects and was configured according to the SEQUENCER protocol configuration from the freely available repository[2], providing non-uniform total order using a sequencer algorithm. Appia is a group communication system that has been used in contributions to STM replication, and was configured with `SequencerUniformLayer` to provide uniform total-order broadcast using a sequencer-based algorithm. Spread uses a privilege-based algorithm and we used the vanilla configuration and message type was set to AGREED, which guarantees non-uniform total order.

5.1 Red-Black Tree

We start by considering a common micro-benchmark from the literature, the Red-Black Tree. This benchmark is composed of three types of transactional operations: (1) insertions, which add an element to the tree (if not already present); (2) deletions, which remove an element from the tree (if present); and (3) searches, which search the tree for a specified element. Insertions and deletions are said to be update transactions. The tree was populated with 32768

[2] `https://github.com/belaban/JGroups`.



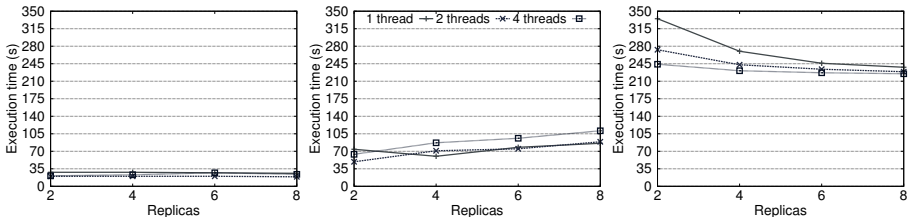
(a) JGroups. (b) Appia. (c) Spread.

Fig. 3. Throughput breakdown in Red-Black Tree, with 2 threads and configured with initial size 32078, range 131072 and 10% updates

pseudo-randomly generated values, ranging from 0 to 131072. Each thread executed 10% of update transactions. Hence, the workload is characterised by very small and fast transactions, with very low contention.

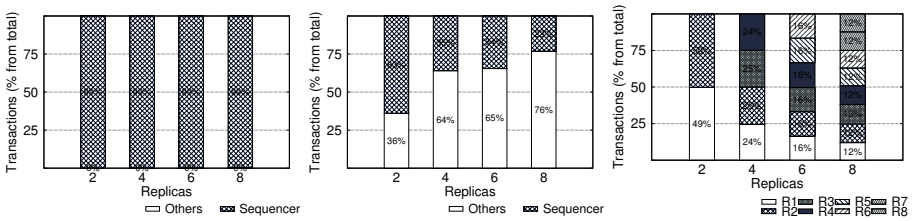
Fig. 2 shows the throughput of our system (in transactions$\times 10^3$ per second, higher is better) on the micro-benchmark, varying the number of replicas and the number of threads per replica. Unsurprisingly, JGroups (Fig. 2a) achieves the best performance of the three for every combination, due to both the fixed sequencer implementation and the uniformity relaxation. Appia (Fig. 2b) quickly peaks at around half the throughput of JGroups, which seems to be hitting a bottleneck, perhaps due to the requirements of the uniform property. Finally, in Fig. 2c, we have the throughput of the system using Spread. The linear scalability displayed is consistent with the idea that the algorithm implemented by Spread achieves fairness. Since every node is provided with equal opportunities to broadcast and order messages, the system scales either with more threads per node, or when nodes increase, not incurring in the bottleneck of a fixed sequencer. In Fig. 2d we have the average total-order broadcast latency for each group communication system. As discussed in §3, the sequencer replica in both JGroups and Appia – JGroups (seq) and Appia (seq), respectively – suffers from lower latencies than the rest of the replicas. With Spread and its privilege protocol the latency is the same for all replicas.

Intuitively, one expects that the higher performance of JGroups is achieved at the cost of unfairness. The sequencer incurs in substantially lower latency, so that replica alone should be dominating the system's throughput. Spread should exhibit the exact opposite behaviour, *i.e.*, each replica contributes evenly to the total throughput. Since the privilege protocol gives exclusive broadcast rights to each replica at a time, hence the same latency for all replicas. In Fig. 3 we breakdown each replica's contribution to the overall system throughput, where the x-axis represents the number of replicas and the y-axis the percentage of transactions each replica executed. Each colour represents the portion of transactions executed by a specific replica. The results corroborate our intuitions. In JGroups (Fig. 3a) the sequencer totally dominates the system's throughput, while in Spread (Fig. 3c) each replica contributes evenly. Appia (Fig. 3b) lies in the middle due to the additional overhead imposed by the uniform property, which is consistent with the measured latencies.



(a) JGroups. (b) Appia. (c) Spread.

Fig. 4. Execution time in Intruder, with the *intruder* configuration from [12]



(a) JGroups. (b) Appia. (c) Spread.

Fig. 5. Transaction breakdown in Intruder, with 2 threads and the *intruder* configuration from [12]

5.2 Intruder

In the Intruder benchmark each thread repeatedly executes 3 phases. The first phase basically involves a simple FIFO queue from which threads pop a packet. In the second phase threads add the packet to a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same flow. If all the packets of a flow have been delivered, they are reassembled and added to the completed packets FIFO queue. The final phase consists of taking a reassembled packet from the FIFO queue and checking if it has been compromised.

The benchmark was parameterized according to the *intruder* configuration in [12]. There were 2048 flows with 4 packets each, and 10 of the flows had been attacked. Transactions under this configuration are small and fast, and the workload is highly contented, due to both of the FIFO queues and the rebalancing of the tree in the reassembly phase. Thus, this workload distinguishes itself from Red-Black Tree's in the contention level.

Fig. 4 shows the execution time (y-axis) when varying the number of replicas (x-axis). The system behaves differently depending on the GCS used. When using JGroups (Fig. 4a) the performance is independent of the number of replicas. The sequencer does all the work and the other replicas have their transactions constantly aborted until the benchmark finishes, as can be seen in the transaction breakdown for JGroups in Fig. 5a. With Appia, Fig. 4b, the system degrades performance as more replicas are added. Since the sequencer does not totally dominate in Appia (Fig. 5b) as in JGroups, this is the expected behaviour due

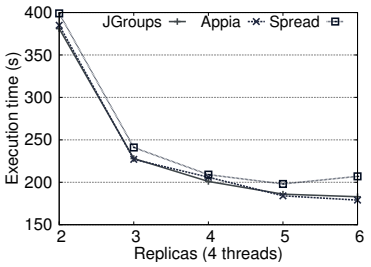


Fig. 6. Execution time in Genome, with the *genome* configuration from [12]

to the contended workload leading to a high abort ratio. With Spread the execution times are much higher (Fig. 4c) due to the privilege-based implementation. Nevertheless, the system performs better when more threads are added.

5.3 Genome

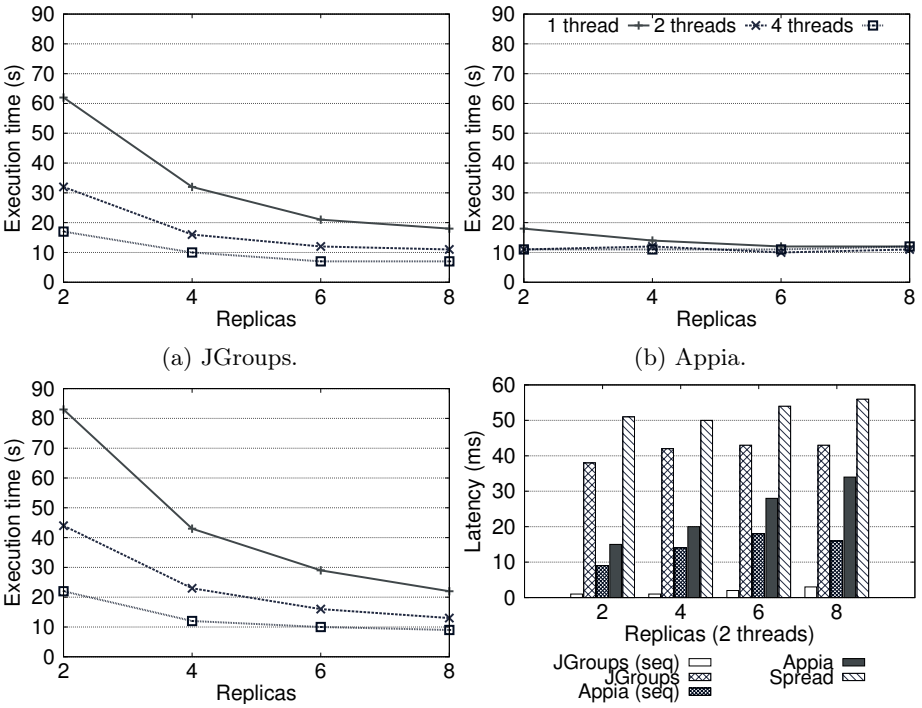
The Genome benchmark consists of several steps which are executed sequentially, but inside each step several threads execute concurrently. But since the steps are sequential, threads wait for each other when advancing from one step to the next. The last step is completely sequential (it is executed by a single thread), and there is one step which is a mix of concurrent and sequential parts.

The benchmark was parameterized according to *genome* configuration in [12]. This workload is radically different from both Red-Black Tree's and Intruder's. Overall, transactions are of moderate length (with regard to the number of operations) and there is little contention. Unlike the previous benchmarks, in Genome data is partitioned among threads. Threads execute a sequence of steps in synchrony, *i.e.*, threads must wait for each other when advancing from step a to step b. With this workload, it is expected that the different total-order broadcast implementations do not have a meaningful impact as replicas can only progress in group. In fact, the system exhibits similar execution times independently of the GCS employed, as seen in Fig. 6.

5.4 Vacation

The Vacation benchmark is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates. Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, transactions are of moderate size.

The benchmark was parameterized according to the *vacation-low* configuration in [12], where contention is low. The database had 16384 records of each reservation item, and clients performed 4096 sessions. Of these sessions, 98% reserved or cancelled items and the remainder created or destroyed items. Sessions



(a) JGroups. (b) Appia.

(c) Spread. (d) Latency.

Fig. 7. Execution time and total-order broadcast latency in Vacation, with the *vacation-low* configuration from [12]

operated on up to 2 items and were performed on 90% of the total records. This workload is similar to Genome's considering that each thread has its own work to perform. Thus, the complete bias of JGroups towards the sequencer should not yield great performance comparing to Spread, since the sequencer can not "steal" the work from the remaining replicas. But unlike Genome, the whole thread execution path is concurrent.

Fig. 7 shows the execution time and latency when executing Vacation with the differents GCS. The most interesting aspect with this experiment is that Appia (Fig. 7b) performs better than both JGroups (Fig. 7a) and Spread (Fig. 7c) right from the start with 2 replicas. Analysing the latencies (Fig. 7d) we can observe that the average latency of Appia replicas is lower than Spread replicas, as expected. With JGroups the sequencer has the usual low latency, but the other replicas exhibit higher latency than any Appia replica. Thus, with Appia replicas can progress concurrently while with JGroups the sequencer finishes first and only afterwards the other replicas progress. Spread exhibits its usual behaviour.

6 Related Work

In this paper we have studied the impact of different total-order broadcast implementations on the regular Non-Voting Certification scheme. The following works are also related to the use of certification-based protocols in STM replication and share the common goal of reducing the coordination overhead, but none studies the impact of the GCS in the system's workload and throughput. Couceiro et al. in [2] propose the use of bloom filters to reduce the size of the messages TO-broadcasted, as the efficiency of the total-order broadcast is known to be strongly affected by the size of the exchanged messages [13]. The authors encode the read set in a bloom filter whose size is computed to ensure that aborts due to the bloom filter's false positives are less than a user-unable threshold. The work in [5] supports the coexistence of the Voting and Non-Voting schemes simultaneously, by relying on machine-learning techniques to determine, on a per transaction basis, the optimal certification strategy to be adopted.

In certification-based protocols transactions are validated at commit time and may be re-executed an unbounded number of times due to conflicts, leading to an undesirably high abort rate. The work in [3] tackles these issues using the concept of lease, informally, a token which gives its holder the privileges to manage a given subset of the whole data set. When certifying a transaction, replicas must first acquire the corresponding leases if not already in possession. Once in possession of the leases, replicas can certify transactions using reliable broadcast instead which is cheaper than total-order broadcast. If a transaction is aborted, the host replica re-executes it without relinquishing the leases.

In [4] the authors exploit the optimistic atomic broadcast primitive in order to reduce the message deliver latency [14]. As soon as a transaction t is optimistically delivered, t is speculatively certified instead of waiting for its final delivery as in conventional certification protocols. This allows an overlap between computation and communication by certifying transactions while the optimistic atomic broadcast computes the final order.

7 Concluding Remarks

STM replication based on certification protocols rely on total-order broadcast. This paper presents, to the best of our knowledge, the first study of the impact that different total-order broadcast implementations can have on several benchmarks used in the literature, each with different workload characteristics. We have found that the same system exhibits different behaviour with regard to performance, fairness and latency, depending on the combination of total-order broadcast implementation and workload characteristics.

These observations open up further work. One can exploit the low latency of the sequencer to schedule update transactions exclusively to the sequencer and read-only transactions (which do not require any remote coordination) to the remaining replicas, which only apply the updates in the background. Additionally, since the remote coordination overhead is very high relative to transaction execution time, and read-only transactions execute exclusively locally, we can execute

read-only transactions while waiting for the delivery of update transactions. This technique is specially appealing for non-sequencer-based implementations, such as Spread's, because latency is higher.

References

1. Shavit, N., Touitou, D.: Software Transactional Memory. In: Symposium on Principles of Distributed Computing (PODC), pp. 204–213 (1995)
2. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D^2STM: Dependable Distributed Software Transactional Memory. In: IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 307–313 (2009)
3. Carvalho, N., Romano, P., Rodrigues, L.: Asynchronous Lease-Based Replication of Software Transactional Memory. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 376–396. Springer, Heidelberg (2010)
4. Carvalho, N., Romano, P., Rodrigues, L.: SCert: Speculative certification in replicated software transactional memories. In: International Systems and Storage Conference (SYSTOR) (2011)
5. Couceiro, M., Romano, P., Rodrigues, L.: PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 309–328. Springer, Heidelberg (2011)
6. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms. ACM Computing Surveys 36(4), 372–421 (2004)
7. Agrawal, D., Alonso, G., El Abbadi, A., Stanoi, I.: Exploiting atomic broadcast in replicated databases. In: European Conference on Parallel and Distributed Computing (Euro-Par), pp. 496–503 (1997)
8. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: International Conference on Distributed Computing Systems (ICDCS), pp. 156–163 (1998)
9. Dias, R.J., Vale, T.M., Lourenço, J.M.: Efficient Support for In-Place Metadata in Transactional Memory. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 589–600. Springer, Heidelberg (2012)
10. Dice, D., Shalev, O., Shavit, N.N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
11. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Poster on the International Conference on Distributed Computing Systems (ICDCS), pp. 707–710 (2001)
12. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE International Symposium on Workload Characterization (IISWC), pp. 35–46 (2008)
13. Kaashoek, M.F., Tanenbaum, A.S.: An Evaluation of the Amoeba Group Communication System. In: International Conference on Distributed Computing Systems (ICDCS), pp. 436–447 (1996)
14. Pedone, F., Schiper, A.: Optimistic atomic broadcast: a pragmatic viewpoint. Theoretical Computer Science 291(1), 79–101 (2003)

How to Cancel a Task

Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer

ETH Zurich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Task parallelism is ubiquitous in modern applications for event-based, distributed, or reactive systems. In this type of programming, the ability to cancel a running task arises as a critical feature. Although there are a variety of cancellation techniques, a comprehensive account of their characteristics is missing. This paper provides a classification of task cancellation patterns, as well as a detailed analysis of their advantages and disadvantages. One promising approach is cooperative cancellation, where threads must be continuously prepared for external cancellation requests. Based on this pattern, we propose an extension of SCOOP, an object-oriented concurrency model.

1 Introduction

Task parallelism has become part of the standard inventory of professional developers, and programming frameworks for this domain are sprouting to help them express their intentions in a safe and concise manner. At the same time, learning to proficiently use such frameworks is far from easy. They offer a confusing variety of abstractions and constructs, often to provide similar but subtly different functionality. Frequently, the only source of information are code examples where the relevance of the constructs cannot be sufficiently discussed. Too little research is spent on consolidating the various approaches by explaining commonalities and differences, which would help developers learn to use new frameworks more quickly and aid designers in developing their frameworks further.

This paper strives to address these deficiencies, focusing on one central problem in task parallelism: task cancellation techniques. Cancellable tasks are mainly used for interrupting long-running or outdated tasks, but the pattern can also be used as a building block for more high-level patterns, such as MapReduce. The paper provides an overview of existing cancellation approaches, extracting techniques from different programming languages and concurrency libraries, classifying them, and discussing their strong and their weak points. This knowledge is then applied to provide a novel task cancellation technique for SCOOP [8,10], an object-oriented concurrency model. The technique is based on the idea of cooperative cancellation where both the canceling and the canceled task must cooperate in order to succeed.

The remainder of the paper is structured as follows. Section 2 provides a taxonomy and discussion of task cancellation techniques. Section 3 describes a cooperative cancellation technique for SCOOP. Section 4 provides an overview of related work and Section 5 concludes with an outlook on future work.

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 61–72, 2013.

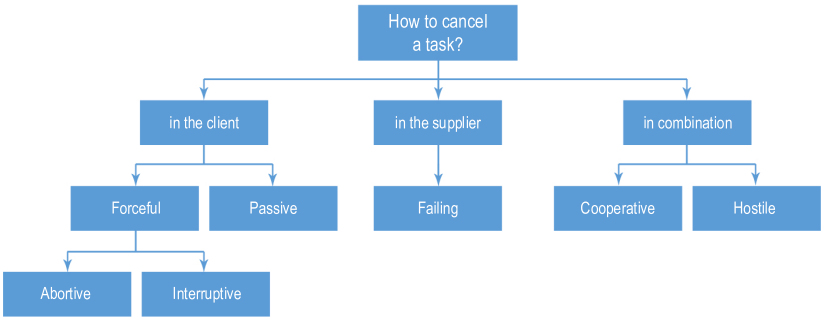


Fig. 1. A taxonomy of task cancellation techniques

2 Classification of Task Cancellation Techniques

A *task* denotes the abstraction of an execution, such as a CPU thread, a thread pool, or a remote machine. Designing a programming model for task parallelism has to deal with the cancellation of tasks, a highly reusable pattern which can be applied to stand-alone applications, client-server systems, and distributed clusters alike. Without proper support for task cancellation, the developer has to write the synchronization code by hand, a task prone to subtle errors.

Various approaches to canceling a running task have been implemented in programming languages and libraries and described in theory. However, so far there is little evaluation and comparison of the proposed techniques. To provide a foundation for discussing them, we have examined a number of popular languages (Java, Python, C# TPL, Pthreads, etc.) and provide a taxonomy in Figure 1.

Client-based cancellation describes techniques where the control over the cancellation process lies entirely with the client (the canceling task):
- *Forceful cancellation* The client forces the supplier (the canceled task) to stop without the possibility to resist:
  - *Abortive cancellation* The supplier is terminated immediately.
  - *Interruptive cancellation* The supplier is allowed to reach a safe point before being terminated.
- *Passive cancellation* The client stops waiting for the result of a supplier, allowing it to continue on its own.

Supplier-based cancellation describes techniques where the control over the cancellation process lies entirely with the supplier:
- *Failing* The supplier encounters an unrecoverable error and needs to inform its clients.

Client/supplier combination describes techniques where client and supplier must act together in order to succeed with the cancellation:
- *Cooperative cancellation* The client asks the supplier to terminate, which decides itself how and when it should terminate.
- *Hostile cancellation* The supplier may resist a cancellation request of the client, and interrupt the client instead.

In the following, we discuss each of the approaches and provide examples of languages where they are employed.

2.1 Client-Based Cancellation

Abortive Cancellation. Immediate termination does not give the canceled thread a chance to respond. As an example, consider the Java code in Listing 1, where `t.stop()` aborts the thread.

```
Thread t = new Thread(){ @override void run(){...} }
t.start();
...
t.stop(); // aborts the running thread
```

Listing 1. Aborting a thread-based task in Java

The advantage of the approach is clearly its simplicity. However, the approach is unsafe because aborting a running thread can leave a program in an inconsistent state. Consider the money transfer example in Listing 2.

```
void transfer(Account from, Account to, int amount) {
  synchronized{
    from.withdraw(amount); // if stopped here, money is lost
    to.deposit(amount);
  }
}
```

Listing 2. Unsafe cancellation using abortion

In this example, a synchronized block is used to guarantee that no thread interferes with the transfer. However, if a running thread is aborted during execution and is forced to unlock all of the monitors that it has locked, the transferred money may be lost and the remaining execution started in an inconsistent state. The Pthreads library [12] with `set_cancellalation mode` set to `PTHREAD_CANCEL_ASYNCHRONOUS` is a further example of abortive cancellation.

Interruptive Cancellation. Using this technique, a running task is aware of potential interruption and usually cannot ignore it. However, a task cannot be canceled in every execution state, but only at so-called *safe points*: places where certain program invariants hold such that the execution may be interrupted safely. Usually a programmer must specify these places by hand, either by calling a library function, or handling a specific type of exception [5]. A special case of this technique allows interrupting a task at only one point in its lifetime: when the task has not been started yet. While it may seem not very useful, in some languages (Scala [13], Python [3]) this is the only built-in cancellation mechanism.

Consider the example in the Pthreads library[1] in Listing 3.

[1] Pthreads supports two cancellation modes: Deferred (as in the example) and Asynchronous. The latter one is an example of *aborting* tasks, with no safety guarantees.

```
// setting the cancellation mode to interruption
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
...
void* CancellablePthread(void* argument){
    ...
    pthread_testcancel(); // the execution can be safely canceled here
}
```

Listing 3. Cancellation points in Pthreads

At a safe point for cancellation, the call `pthread_testcancel`() checks on potential cancellation requests. Additionally, some of the blocking system calls are also considered to be cancellation points in Pthreads [12]. Java's thread interruptions[2] and `thread.Abort`() in C# (unlike the method's name is suggesting) are another examples of interruptive cancellation [6,1].

Its potential safety guarantees are a benefit of this approach: *if* the approach is applied correctly, a program can be considered to be in a consistent state after a cancellation. Writing correct interruption-aware code is however difficult [11,1] as a programmer has to remember subtle rules (e.g. in Pthreads some I/O calls are interruptible, others are not) and maintain a program's invariants by hand.

Passive Cancellation. This technique is different from the forceful methods in that a canceling task does not need to become active: it simply stops waiting for a task result, while the running task is still being executed.

As an example, consider downloading a file over a network, illustrated in Listing 4 with C#'s Task Parallel Library (TPL). The call to a `StartDownload`() is asynchronous and returns only a handle to a future (an object, representing a computation that is still being computed [2]), represented by the `Task` class. After some time the downloader's result might not be needed anymore, i.e. the execution is abandoned (in the **if** branch).

```
void PassiveCancellation(string url){
    Downloader downloader = new Downloader(url);
    Task<byte[]> bytesFuture = downloader.StartDownload();
    ...
    if(noNeedToDownload) {
        // the download is not needed anymore
        return; // data is still being downloaded ...
    }
    else {
        // the download is still needed
        var result = bytesFuture.Result; // fetching the result
    }
}
```

Listing 4. Passive cancellation in the Task Parallel Library (TPL) of C#

Obviously, this approach is not uniformly applicable; for example, we might still want to cancel a state-changing procedure. And it is important to know

[2] User-defined code may ignore interruption [11], but only between calls to library methods (which will not ignore it).

in advance that the task will eventually be completed, i.e. listening to a TCP-socket cannot be canceled in this way. Another disadvantage is that the running task continues to consume machine resources. However, in a distributed setting this approach can find its application: consider a framework for a distributed computing, such as MapReduce. Often for the last piece of work several tasks are spawned [4] but only a single result will be used. In this case, there is no need to write sophisticated cancellation code, and it is valid to "forget" about the remaining executing tasks.

2.2 Supplier-Based Cancellation

This class of techniques deals with the special case that cancellation is not requested by a client but that a failure happens in supplier, i.e. it cannot fulfill it's obligations to clients; the supplier therefore needs to terminate. To indicate a failure, exceptions are typically used in object-oriented programming environments. Hence, this case boils down to the problem of exception handling in concurrent environments [9], which is not the focus of this paper.

2.3 Client/Supplier Combination

Cooperative Cancellation. A gentle way to stop a task is to cooperate and *ask* it to do so. The rationale for this approach is simple: a task is the abstraction of an execution, and hence should contain the information about how and when it should be stopped.

In other words, a task must be ready to be canceled at any time by external request. C#'s Task Parallel Library (TPL) follows this pattern, where a single point of cooperation is denoted by two classes: `CancellationTokenSource`, a generator of `CancellationToken`, which itself is a concrete request to cease the execution. An example is given in Listing 5.

```
void Client(){
    var cts = new CancellationTokenSource();     // create the token
        source
    // pass the token to the cancelable operation
        Task.Run(() => Supplier(cts.Token));
        ...
    cts.Cancel(); // request cancellation
}
void Supplier(CancellationToken token) {
    for (int i = 0; i < 100000; i++) {
                // some work
        if (token.IsCancellationRequested) {
            break; // potentially perform cleanup, terminate
        }
    }
}
```

Listing 5. Cooperative cancellation in TPL

This technique provides a solid general structure for writing a cancellable tasks (see Listing 6), with a guarantee that no invariants will be violated. Unlike

Table 1. Dueling rules

	retain	*yield*
demand	the client is interrupted	the supplier is interrupted
insist	the client waits	the supplier is interrupted

in interruptive cancellation, the programmer does not need to remember subtle rules of a concrete library or language. Cooperative cancellation also does not require any runtime support. Unfortunately, one cannot use the true power of this technique unless libraries support this pattern too (as far as we know, to date only limited support is introduced in C#). As another disadvantage, the latency between a cancellation request and actual cancellation is increased.

```
void function run(CancelRequest cancel)
    while(not is_done){
        if cancel is requested
            exit
        loop_once
    }
//need to be specified for concrete task.
void function loop_once;
boolean function is_done;
```

Listing 6. General structure of a cancellable task in cooperative cancellation

Hostile Cancellation. While in the previous paragraph client and supplier are cooperating in order to succeed, in hostile techniques involve a struggle between the canceling and the canceled task. We describe these techniques on the example of *duels*, a mechanism was theoretically described in [8] but not yet implemented. We are using the terminology from [8] in the rest of this chapter.

The key insight in this approach is that a canceling task (a "challenger", in the original) might not be strong enough to request an actual cancellation. If the canceling task is worthy enough, its request is fulfilled (the task is "killed"); if not, it gets an exception itself (therefore the approach is named a duel). In other words, dueling is a two-way interruption, where the result depends on which of the tasks is stronger.

To specify its preferences, a supplier can be in one of the two modes: either *retain* or *yield*. The former means that the task refuses to be canceled, and the latter specifies that it is OK to be interrupted. On the side of the client, there are also two options available: *demand* and *insist*. The first is more impatient, the second more gentle. The complete set of rules is shown in Table 1.

The dueling mechanism is useful in environments where executions are prioritized. For example, one can imagine a robotics system that needs to handle simultaneously a variety of different tasks: route planning, controlling the motors, etc. These tasks can arise non-deterministically and compete for processing units, attempting to cancel other activities. However, it is completely unacceptable that low priority task succeeds in canceling a more important one

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER [INTEGER]

    consume (a_buffer: separate BUFFER [INTEGER])
  -- consume an item from the buffer
  require
    not (a_buffer.count = 0)
  local
    consumed_item: INTEGER
  do
    consumed_item := a_buffer.item
  end
```

Listing 7. Producer-consumer example in SCOOP

(for example, the data collection routine should not be able to cancel a task that adjusts the speed of the motors). In this case, a proper setup of dueling rules could both permit a cancellation request from high priority challengers and provide security from cancellation for important computational tasks.

3 Cooperative Cancellation in SCOOP

This section provides an introduction to SCOOP, an object-oriented concurrency model for contract-equipped languages, and evaluates the patterns of Section 2 for use within this model. Furthermore, the section shows how the cooperative cancellation pattern can be applied to a SCOOP, implemented in Eiffel. For the rest of the paper, we are using Eiffel notation and terminology [8].

3.1 Overview of SCOOP

The goal of SCOOP [8,10] is to provide a simple and safe way to write concurrent code while retaining sequential object-oriented programming principles in as far as possible. Each object in SCOOP is associated with a *processor* (typically implemented as a thread), called its *handler*. Features of objects that reside on different processors can be executed in parallel. The keyword **separate** is used to mark objects residing on different processors, relative to the current object.

The producer-consumer problem serves as an illustration of these ideas. The main entities `producer`, `consumer`, and `buffer` are shown in Listing 7. The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one. A *creation instruction* on a separate entity such as `producer` will create an object on another processor; by default the instruction also creates that processor.

A consumer accesses an unbounded buffer in a feature call `a_buffer.item`. To ensure exclusive access, the consumer must lock the buffer before accessing it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature

execution, thus preventing data races. Such targets are called *controlled*. For instance, in `consume`, `a_buffer` is a formal argument; the consumer has exclusive access to the buffer while executing `consume`.

Condition synchronization relies on preconditions (after the **require** keyword) to express wait conditions. Any precondition makes the execution of the feature wait until the condition is true. For example, the precondition of `consume` delays the execution until the buffer is not empty.

3.2 Choosing a Cancellation Mechanism for SCOOP

The main goal of SCOOP is to provide an easy-to-use model for expressing concurrency, with a focus on the correctness of the resulting programs. Any cancellation mechanism proposed for SCOOP must be designed in this spirit.

Clearly, *abortive cancellation* is an error-prone pattern, and it does not go well with SCOOP's focus on design-by-contract mechanisms. *Interruptive cancellation* has no strict correctness guarantees and can be complicated to use, which does not correspond to SCOOP's simplicity principle. In other concurrency models, where stricter techniques are not favored, interrupting may be a viable option. As mentioned, *passive cancellation* does not need to be explicitly implemented. As it is highly depended on particular usage scenarios, and has no guarantees that it will succeed (the termination of a passively canceled thread is not ensured), it also partly contradicts SCOOP design principles.

A concurrent object-oriented language needs to have well-defined rules about exception handling. The SCOOP implementation is discussed in [9].

As a simple and safe approach, *cooperative cancellation* is a natural candidate to be implemented in SCOOP. It can be implemented as a library approach, thus even eliminating the need to modify the compiler. *Dueling* could be considered as an alternative to cooperative cancellation. However, while duels are only a good fit for specific scenarios, and less suitable in others, we prefer cooperative cancellation as a general-purpose approach.

3.3 SCOOP with Cooperative Cancellation

It is instructive to try to directly implement the approach introduced in Listing 6 in SCOOP. One can start with an abstract[3] class `CANCELLABLE_EXECUTOR` and introduce a `CANCEL_REQUEST` as a shared object that propagates a cancellation request; the descendants need to define the termination criteria in `is_done` and a single loop iteration `loop_once`.

An example of using this implementation of cooperative cancellation is shown in Listing 8. Unfortunately, this attempt does not work because in SCOOP a cross-processor call of `run` would force the executing processor to block on executing the loop for the entire execution. Thus all subsequent cancellation requests would be queued in the processor's request queue, effectively making `CANCELLABLE_EXECUTOR` useless. This happens because `CANCELLABLE_EXECUTOR` is both responsible for listening to cancellation requests and the execution itself.

[3] **deferred** in Eiffel notation.

```
executor: separate CONCRETE_CANCELLABLE_EXECUTOR
cancel: separate CANCEL_REQUEST --shared between two processors
...
-- launching an execution
executor.run (cancel) -- execution started on different processor

cancel.request -- canceling an execution
```

Listing 8. Usage of `CANCELLABLE_EXECUTOR`

This problem can be solved by decoupling the listening and the execution logic; the design is provided in Figure 2. The `CANCELLABLE_EXECUTOR` is now responsible only for listening for cancellation requests; the actual execution is handled by a different processor. To represent a concrete execution, a concretization of the deferred class `EXECUTION_UNIT` is needed. The `CANCEL_REQUEST` may not be actually **separate**, but keeping it this way provides additional flexibility for the case when cancellation request is coming from a client residing on processor separate from `CANCELLABLE_EXECUTOR`.

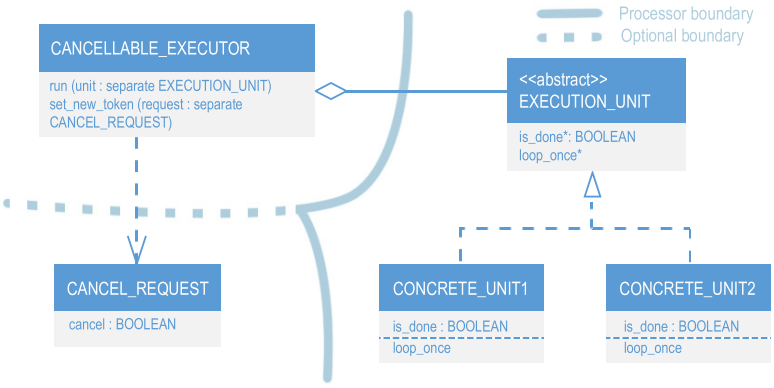


Fig. 2. SCOOP cancellation design

In this design, `EXECUTION_UNIT` is a deferred class, only responsible for performing a single-loop iteration and termination criteria[4]. A task's life cycle is expressed in `CANCELLABLE_EXECUTOR` (see Listing 9), with the following methods:

- `make` (omitted) creates an empty cancellation request. At this point execution cannot be canceled from the outside.
- `set_new_token(a_token: separate CANCELLATION_REQUEST)` sets a new cancellation request, enabling a cancellation.
- `run(a_unit: separate EXECUTION_UNIT)` accepts the execution unit (where execution details are encapsulated) and starts the cancellation-aware execution, according to Listing 6.

[4] This functionality could also be implemented with Eiffel *agents* (function objects), but we present an abstract class to avoid providing execution details.

```
class    CANCELLABLE_EXECUTOR
feature
    set_new_token (a_token: separate CANCELLATION_REQUEST)
    do token := a_token end

    run (a_unit: separate EXECUTION_UNIT)
    do
        from until
            is_done (a_unit) or cancel_requested
        loop
            if check_cancel_requested (token) then
                cancel_requested := TRUE
            else
                loop_once (a_unit)
            end
        end
        cancel_requested := FALSE
    end
end

feature{NONE}
    cancel_requested: BOOLEAN
    token: separate CANCELLATION_REQUEST
```

Listing 9. Cancellable executor

As soon as a cancellation is requested, the loop body can be executed at most once more. After one cancellation request, the instance of `CANCEL_REQUEST` becomes useless, therefore we provide `set_new_token` to refresh a request as many times as needed.

3.4 Example of Usage

As an example of using cooperative cancellation in SCOOP, we present a downloader application that requests a URL, starts a background download process and provides progress reports. While still in progress, downloading can be canceled by the user. The complete source code is available for download;[5] in the following description, we focus on key aspects of this application.

The `DOWNLOADER_UNIT`, responsible for downloading a single portion of bytes from specified URL, is shown in Listing 10 (some code is omitted for brevity). Note that a **separate STRING** is required in the constructor to obtain control over it, as `DOWNLOADER_UNIT` resides on a different processor than its clients. The implementation is straightforward otherwise. Launching an asynchronous download task is done in a pattern similar to Listing 8, applying the design in Section 3.3. One should create one `CANCEL_REQUEST` per launch: the cancel requests are designed to be used only once.

4 Related Work

To the best of our knowledge, this work is the first to attempt a comprehensive classification and evaluation of task cancellation techniques. The work closest

[5] `http://se.inf.ethz.ch/people/kolesnichenko/src/downloader_sample.7z`

```
class DOWNLOADER_UNIT inherit EXECUTION_UNIT
feature
    make (a_url: separate STRING)
    do
        -- use URL 'a_url' and init http_downloader (omitted)
        create parts.make -- create a storage buffer
    end

    is_done: BOOLEAN -- done when all bytes are transferred
    do
        Result := http_downloder.bytes_transferred = http_downloder.
            count
    end

    action
    do
        http_downloder.read
        if attached http_downloder.last_packet as last_p then
            parts.put_front (last_p)
        end
    end

feature {NONE}
    http_downloder: HTTP_PROTOCOL
    parts: LINKED_LIST [STRING] -- buffer for content
end
```

Listing 10. Example: Downloader unit

to ours is [11], Chapter 7, where some cancellation techniques are discussed for Java. In particular, cooperative cancellation with a shared variable or future is presented, along with rules to write a correct interrupt-aware code.

Further related work is also found in descriptions of individual techniques as part of language and library designs. The degree of support of task cancellation varies in such approaches. C# natively supports interruptive cancellation, and since release of TPL also cooperative techniques were introduced [7].

Things get even more complicated when cancellation involves several tasks that need to agree on shutting down and terminate in a safe order. One approach taking this into account is applied to OpenMP [14]. The authors introduce an abortive cancellation of already launched OpenMP tasks (which boils down to the cancellation mechanisms of Pthreads), with unrestricted possibility to cancel unstarted tasks. Their techique works on task groups, involving a child-parent relationship allowing to cancel the whole group, starting from the root.

Python supports interruptive cancellation of non-started tasks via executors [3] and abortive cancellation of already started ones. Similiarly, Scala supports the `Cancellable` interface which allows canceling only non-started tasks [13].

Java supports interruptive cancellations natively [11]. Pthreads library supports both abortion and interruption, depending on the setup [12].

5 Conclusion

The role of parallel programming in modern applications is notable and continuously increasing; research into programming models for this setting is therefore

critical. Indeed, many new models and frameworks for concurrent and parallel programming have been proposed in the past decade. To provide guidance in this vast field, it is important to consolidate the knowledge about commonly used patterns, and to provide a coherent frame for discussion and evaluation. In this paper we selected one important task parallelism pattern, task cancellation, provided a taxonomy of techniques, scrutinized their usage, and proposed a novel cancellation technique for SCOOP on this basis.

In future work, cooperative cancellation in SCOOP could be further extended to support task chaining (canceling an intermediate task causes all other tasks to be canceled) and precondition-aware tasks (these could effectively be deferred in the executing process). Another extension of this work is to provide a formal model, describing the control flow in different cancellation techniques.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA).

References

1. Albahari, J., Albahari, B.: C# 3.0 in a Nutshell: A Desktop Quick Reference. O'Reilly Media, Incorporated (2007)
2. Baker Jr, H.C., Hewitt, C.: The incremental garbage collection of processes. In: Artificial Intelligence and Programming Languages, pp. 55–59. ACM (1977)
3. Concurrent futures in Python (2013), `http://docs.python.org/dev/library/concurrent.futures.html`
4. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
5. Destroying Threads in C# (2013), `http://msdn.microsoft.com/en-us/library/cyayh29d.aspx`
6. Hyde, P.: Java thread programming. Sams Pub. (1999)
7. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. In: OOPSLA 2009, pp. 227–242. ACM (2009)
8. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)
9. Morandi, B., Nanz, S., Meyer, B.: Who is accountable for asynchronous exceptions? In: APSEC 2012, pp. 462–471. IEEE Computer Society (2012)
10. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. Ph.D. thesis, ETH Zurich (2007)
11. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley (2005)
12. POSIX threads specification (2013), `http://man7.org/linux/man-pages/man7/pthreads.7.html`
13. Scala Scheduler (2013), `http://doc.akka.io/docs/akka/snapshot/scala/scheduler.html`
14. Tahan, O., Brorsson, M., Shawky, M.: Introducing task cancellation to openMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 73–87. Springer, Heidelberg (2012)

Automatically Repairing Concurrency Bugs with ARC

David Kelk, Kevin Jalbert, and Jeremy S. Bradbury

Software Quality Research Laboratory
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
{david.kelk,kevin.jalbert,jeremy.bradbury}@uoit.ca

Abstract. In this paper we introduce ARC – a fully automated system for repairing deadlocks and data races in concurrent Java programs. ARC consists of two phases: (1) a bug repair phase and (2) an optimization phase. In the first phase, ARC uses a genetic algorithm without crossover to mutate an incorrect program, searching for a variant of the original program that fixes the deadlocks and data races. As this first phase may introduce unneeded synchronization that can negatively affect performance, a second phase attempts to optimize the concurrent source code by removing any excess synchronization without sacrificing program correctness. We describe both phases of our approach and report on our results.

Keywords: bug repair, concurrency, concurrency testing, evolutionary algorithm, SBSE.

1 Introduction

As computers and even mobile devices now ship with more than one core per chip, programs must parallelize to take advantage of improvements in processing power [23]. On a multicore system concurrency provides a potentially significant benefit with respect to performance, however, writing concurrent source code can be difficult and error-prone, especially when one considers the set of possible thread interleavings of a concurrent program. Further exacerbating the issue of writing correct concurrent source code is the fact that concurrency bugs can be difficult to find because they may occur in only a small set of possible thread interleavings [21]. Even when a concurrency bug has been detected, its repair is often non-trivial because many concurrency bugs are the result of the interaction of different code fragments executing in different threads within a program.

The use of search-based software engineering (SBSE) [12] techniques to automatically repair bugs in sequential programs is a well researched idea [17,24]. To address the challenges of detecting and fixing concurrent programs we propose ARC (**A**utomatic **R**epair of **C**oncurrency bugs) – an automatic technique to repair deadlocks and data races in concurrent Java programs. ARC requires no

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 73–84, 2013.

formal specifications or annotations. Only the Java source code and a test suite capable of demonstrating known deadlocks and data races are necessary. ARC consists of two phases: (1) a bug repair phase and (2) an optimization phase. At its core, ARC works by using a genetic algorithm without crossover ($GA\neg C$) to evolve variants of an incorrect concurrent Java program into a variant that fixes all known bugs.

A common problem for automatic bug repair techniques is the size of the search space of possible bug fixes. Applying these techniques to concurrent programs introduces thread interleavings which increase the difficulty of searching the space. To counteract this challenge, ARC incorporates techniques to constrain the search space and make it tractable. First, we limit the algorithm to only fixing deadlocks and data races in concurrent Java programs (instead of all types of concurrency bugs). Second, ARC only targets modifications to concurrency mechanisms as potential bug fixes. For example, `Synchronize` statements maybe added, removed, and manipulated. The possible bug fixes are comprised of the combined application of 12 mutation operators to evolve the program[1]. Third, the Chord tool [22] is used to perform a static analysis of a concurrent program to target specific shared classes, methods and variables where bug fixes can be applied (i.e., localization of bug fixes within the source code). This shared list is further refined by the ConTest testing tool [7]. Fourth, the evaluation of potential bug fixes are evaluated using ConTest which injects noise (i.e., random delays) that assist in exploring the interleaving space during testing.

Next in Section 2 we cover background material related to concurrency bugs and genetic algorithms. The motivation for ARC along with an example problem are presented in Section 3. In Section 4 we describes how ARC evolves fixes for data races and deadlocks (Phase 1). Improving nonfunctional properties of the program, such as its execution time (Phase 2), is described in Section 5. We evaluate ARC in Section 6 on a set of programs from the IBM concurrency benchmark [8, 9, 13]. In Section 7 we discuss related works in the field of automatic program repair and explain how ARC is novel when compared to previous approaches. Finally, we conclude and present future work in Section 8.

2 Background

2.1 Concurrency Bugs

Data races and deadlocks are two of the most common concurrency bugs. A data race can be defined as: *"... two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [20]. A deadlock can be defined as: *"... a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ... For example, a deadlock can occur when one thread in a program holds a lock that another thread desires and vice-versa"* [20].

[1] A number of the mutation operators used to repair bugs in ARC are converse to the mutation operators in the ConMAn mutation testing tool [4].

Buggy Program:

```
write(int var1){
  ... // Expensive loop
  data = var1;
  ... // Database query
}

int public read(){
  return data;
}
```

Fixed Program:

```
synchronize write(int var1){
  ... // Expensive loop
  data = var1;
  ... // Database query
}

int synchronize read(){
  return data;
}
```

Fig. 1. A developer first synchronizes the `read` function, yet the bug still exists. Synchronizing the `write` method as well fixes the bug.

1^{st} **Optimization on Fix:**

```
public write(int var1){
 ... // Expensive loop
 synchronized(this){
   data = var1;
   ... // Database query
 }
}

int synchronize read(){
 return data;
}
```

2^{nd} **Optimization on Fix:**

```
public write(int var1){
 ... // Expensive loop
 synchronized(this){
   data = var1;
 }
 ... // Database query
}

int synchronize read(){
 return data;
}
```

Fig. 2. A developer shrinks the critical region to exclude the expensive loop (Optimization 1). Next, a developer shrink the critical region again to exclude the database query (Optimization 2).

2.2 Genetic Algorithms

Genetic algorithms [11] (GAs) are a heuristic search technique modelled on natural evolution. They are population based and uses mutation, crossover and a fitness function to evolve solutions to problems. Proposed solutions are encoded in individuals of the population. Each individual is evaluated by a fitness function, an equation that determines how close the individual is to the solving the problem. The more fit a individual's solution is, the greater the chance it will pass it's genetic material (i.e., itself) into the next generation. Crossover mixes the individuals to produce new ones while mutation injects fresh information in to the population so it does not become stagnant.

ARC uses a genetic algorithm without crossover (GA$\neg$C) and selection. A population of proposed solutions is generated in the first generation and mutated each generation. Two ending conditions exist. First, a fix is found for the data races and deadlocks or a fixed number of generations pass with no solution found.

3 Motivating Example

To illustrate the challenges of concurrency bug repair we consider an example of a data race and how ARC might fix it. In the left part of Figure 1 the `read` and `write` method access a shared variable. A very simple data race exists because there is no atomic access to the `data` variable during the concurrent reading or writing. A possible repair involves synchronizing both accesses as shown in the right part of Figure 1. Note that synchronizing one method alone does not fix this bug. Section 4 describes this in more detail.

The solution in the right part of Figure 1 is far from ideal. The solution found by ARC forces other threads to wait unnecessarily while the write method works in the loop and database sections. An optimization is to shrink the critical region guarded by the synchronize statements to only guard access to the shared variable as shown in Figure 2. Section 5 describes how ARC attempts to optimize fixes found in the first phase of operation by removing and shrinking synchronization blocks.

4 Phase 1: Fixing Deadlocks and Data Races

ARC requires two inputs: An incorrect concurrent Java program and JUnit tests exercizing the errors. The test suite is the oracle that determines if bugs still exists in the program. One limitation of ARC (and of other related automatic bug fixing techniques mentioned in Section 7) is that it can only fix bugs detectable by the test suite. Given an incorrect program ARC performs a static analysis to identify the variables, classes and methods involved in concurrency. It then invokes the GA¬C to find fixes for the data races and deadlocks. Each generation is broken down into a number of steps, shown in Figure 3 and described here.

Update Population. First, the members of the population must be created. If ARC has just started, the original incorrect program is replicated and copied into each member of the GA¬C in the first generation. For succeeding generations N the program for the same member from generation N - 1 is used.

Generate Mutants (Update Representation). After the population is updated, ARC generates all mutants for all members using the operators in Table 1. These operators are implemented in TXL [6] using pattern matching and replacement rules. An example mutant created by the EXSB operator is shown in Figure 4.

Apply a Mutation to an Individual. Once all mutants are generated for each individual, ARC selects a type of mutation (e.g. EXSB) and then an instance of it (e.g. 4^{th} mutant generated) from those available. The selected mutation is copied into the source for the member. It is possible that the mutant is not valid. For example, a new synchronization block could have been added that synchronizes on a variable that is out of scope. ARC attempts to compile the project. If an error is detected, the mutation is rolled back and another is selected. This continues until a successful compilation or ARC runs out of mutants. In this latter case, ARC raises and exception and ends.

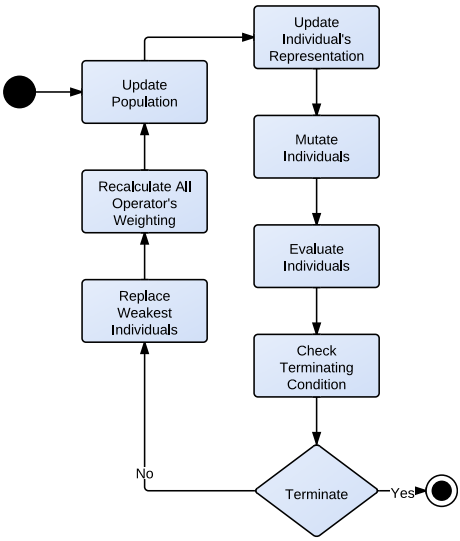


Fig. 3. Detailed view of the GA¬C used in Phase 1, the repair phase

Table 1. Set of mutation operators used by ARC

Operator Description	Acronym
Add a synchronized block around a statement	ASAT
Add the synchronized keyword to the method header	ASIM
Add a synchronized block around a method	ASM
Change the order of two synchronized blocks order	CSO
Expand synchronized region after	EXSA
Expand synchronized region before	EXSB
Remove synchronized statement around a synchronized statement	RSAS
Remove synchronization around a variable	RSAV
Remove synchronized keyword in method header	RSIM
Remove synchronization block around method	RSM
Shrink synchronization block after	SHSA
Shrink synchronization block before	SHSB

Program P:

```
obj.write(var1);
synchronized(lock){
  myHash.remove(var1);
}
```

Program P':

```
synchronized(lock){
  obj.write(var1);
  myHash.remove(var1);
}
```

Fig. 4. An example of the EXSB (expand synchronization before) mutation operator

Evaluate Individuals. A mutation may be beneficial, destructive or benign. We must evaluate it to determine it's effect on the program. A key problem

in evaluating mutants is the unpredictability of thread interleavings. If a concurrency bug appears in only a few possible interleavings, how can we gain confidence that a proposed fix actually works?

ARC uses IBM's ConTest tool [7] to instrument the software under repair by injecting noise into the selection of interleavings. This causes threads to randomly delay at different times during execution, increasing the chance that different interleavings are explored. By running the instrumented version of the program multiple times we gain more confidence that a larger set of the interleavings are explored. Choosing the number of times ConTest run to test each proposed fix is the most crucial parameter in ARC. Confidence in any proposed fix must be carefully balanced against the time required to find the fix.

The number of successful ConTest executions are used to determine fitness:

$$functional\ fitness(P) = (s \times sw) + (t \times tw)$$

$$where:\ s = \#\ of\ successful\ executions,\ sw = success\ weighting,$$
$$t = \#\ of\ timeout\ executions,\ tw = timeout\ weighting$$

When determining fitness we consider both a successful execution and a timeout as positive factors. Timeout executions are positive because given more time they may become successful executions. Timeouts are weighted less than a successful execution since the success of a timeout execution is not guaranteed.

If ARC finds an individual that achieves 100% successful executions, we need to ensure it is truly a fix. It is possible that a proposed solution could still contain a bug that escaped detection because the interleaving exhibiting it were not selected by ConTest. To increase our confidence that a fix is correct we take the base number of ConTest runs (say 10) and multiply it by an additional safety factor (say 20). We run the proposed fix through ConTest ($10 \times 20 = 200$) more times to give us additional confidence the fix holds. If a data race or deadlock is found during these additional runs, the fix is rejected and the search continues. This continues until a correct program is found or the GA¬C runs out of generations.

Replace Weakest Individuals. We believe the *competent programmer hypothesis* [1] applies when fixing concurrency errors. That is, that programmers strive to create correct programs. Programs with bugs in them are nearly correct so the distance in the search space from an incorrect program to a correct program is small(er) and tractable. Even with a smaller search space, evolutionary algorithms may evolve candidate solutions that stray down paths leading to little or no improvement. To encourage individuals to explore more fruitful areas of the state space we include the option to restart or replace the lower w percentage (say 10%) of individuals if they under-perform for too long. Two replacement strategies are used. First, the under-performing member is replaced with a random individual from the upper x percent of the population. Second, the member is replaced by the original incorrect program.

Recalculate Operator Weighting. ARC leverages historical information on how successful different mutation operators have been and about the relative dominance of data races and deadlocks. We give additional weight to operators

that have raised the fitness of the population or reduced the frequency[2] of data races or deadlocks. The weighting is designed to ensure the chance of selecting an operator is always greater than zero regardless of their performance. Separate weightings are used for data races and deadlocks. A sliding window of n generations is used to adapt the operator weighting to recent history.

5 Phase 2: Optimizing Fixes from Phase 1

ARC may introduce unnecessary synchronization during the fixing process. If a fix is found, an optional second phase begins that attempts to improve the running time of ARC by shrinking and removing unnecessary synchronization blocks. The same strategy is used from part one. A new fitness function and a subset of the TXL operators (RSAS, SHSA, etc. from Table 1) are used.

$$non-functional\ fitness(P) = \frac{worst\ score}{[sig_t \times unc(t)]+[sig_c \times unc(c)]}$$

$$where:\ unc(x) = \frac{(x_{max}-x_{min})}{x_{avg}}$$

$$sig_t = \begin{cases} t/c & if\ t\ > c \\ c/t & if\ c\ > t \end{cases}$$

$$sig_s = \begin{cases} c/t & if\ t\ > c \\ t/c & if\ c\ > t \end{cases}$$

The fitness function depends on the real time, t, required for an execution and the number of voluntary context switches made, c. Voluntary context switches is the number of times a thread voluntarily gives up control of the CPU. By minimizing unnecessary synchronization both of these values should decrease. At the beginning of phase 2 we run the correct, unoptimized program (without ConTest) a number of times to acquire the unoptimized running time and number of context switches. These values are used in the fitness function to evaluate relative improvements.

Removing and reducing synchronization runs the risk of introducing new errors into the program. Before every non-functional evaluation we need to ensure that no bugs are present. We re-run the first phase's correctness check (eg, 10 ConTest runs, then 200 more). If any deadlocks or data races are encountered the proposed optimization is rejected and this individual is reset to the previous generation. After ARC validates the proposed optimization additional runs are conducted without using ConTest to obtain the running time and voluntary context fixes.

Unlike phase 1 there is no early stopping criteria as there is no correct running time. Lower is always better. Phase 2 always uses its full allotment of generations. For this reason, running phase 2 is user-configurable. Optimization of this phase is future work.

[2] For example, if EXSB reduces the occurrence of deadlocks from 80 of 100 ConTest runs to (say) 60 of 100 runs, it will be selected more frequently in future generations to combat deadlocks.

Table 2. The set of IBM benchmark programs used to evaluate ARC

Program	SLOC	Classes	Bug Type	Can Fix?
account	165	3	Data Race	Yes
accounts	75	2	Data Race	Yes
bubblesort2	104	2	Data Race	Yes
deadlock	109	2	Deadlock	Yes
lottery	157	2	Data Race	Yes
pingpong	143	4	Data Race	Yes
airline	93	1	Data Race	No
buffer	319	5	Data Race	No

6 Evaluation

In order to evaluate ARC's ability to repair concurrency bugs we selected 8 programs from the IBM Concurrency Benchmark [8,9,13]. We chose six programs containing bugs ARC can fix and two ARC cannot fix as a sanity check.

ARC is designed to be flexible and contains a number of configurable parameters. Table 3 describes the configuration used in our evaluation. The parameter values are influenced by community standards (e.g., evolution population, evolution generations) and through experience gained using ARC (e.g., ConTest runs, validation multiplier). Of importance, the GA¬C population size and generation size are both 30. Every member at every generation is evaluated by being run through ConTest 10 times. Any potential fix is evaluated 150 more times. Testing was conducted on a Linux PC with a 2.33 GHz processor, 4 gigabytes of RAM running Linux Mint 13.

6.1 Experimental Results

Each program in Table 2 was run through ARC 5 times using the parameters described in Table 3. Results are summarized in Table 4. ARC was able to fix the 6 fixable programs and was not able to fix the 2 non-fixable. For the 6 repairable programs the time taken to find a fix ranged from about 2 minutes to 100 minutes. The most time consuming aspect of ARC is the numerous ConTest executions. Second is the waiting necessary to determine the difference between a successful execution and a timeout caused by a deadlock. The *Timeout Multiplier* in Table 3 allows ARC to wait up to 20 times the instrumented execution time for the program to complete.

Almost all fixes are found in the first or second generation. The static analysis by Chord and the dynamic analysis by ConTest significantly shrink the state space. For example, the account program contains 3 classes, approx. 9 methods and 6 variables. After the analysis, this is reduced to 2 classes, 3 methods and 3 variables. A population of 30 may exceed the number of mutations available, leading to the search space probably being exhaustively covered. If the correct program is 1 or 2 mutation steps from the incorrect one, it should be found

Table 3. The set of parameters that ARC uses along with their descriptions and values

Parameter	Description	Value
Project Test MB	The amount of memory allocated	2000
ConTest Runs	Test suite executions per gen. per member	10
Validation Mult.	Multiplier on ConTest runs when validating potentially correct programs	15
Timeout Mult.	Time multiplier for ConTest before timeout	20
Evolution Gen	Maximum number of generations of the GA¬C	30
Evolution Population	Population size for the GA¬C	30
Replace Lowest %	Lowest n% of population replaced in GA¬C	10
Replace With Best %	Replace under-perfomers with best individuals n% of the time	75
Replace min turns	Minimum time under-performing	3
Replace Interval	Every n generations, under-performers are replaced	5
Ranking Window	Size of sliding window for operator weighting	5
Success Weight	Fitness score for successful executions	100
Timeout Weight	Fitness score for timeout executions	50
Improv. Window	Number of generations to consider for convergence	10
Avg. Fit. Delta	Minimum average fitness improvement required	0.01
Best Fit. Delta	Minimum best fitness improvement required	1

Table 4. Summary of the results of running the programs through ARC 5 times

Program	Average Generations to Find Fix	Average Time Taken (HH:MM:SS)
account	5.0	00:08:08
accounts	1.0	00:44:00
bubblesort2	2.2	01:40:20
deadlock	1.0	00:02:12
lottery	2.4	00:38:00
pingpong	1.0	00:12:32

quickly. ARC works as a proof of concept and must be further evaluated – more runs and on larger programs.

Threats to Validity. The main threat to validity for our experimental evaluation of ARC is external validity – our ability to generalize results. All of the programs used in our experiment are small and are not representative of large-scale concurrent software. In the future, we plan to address this threat by conducting further experiments with larger concurrent software systems.

7 Related Work

We will now discuss two areas of related research: the use of SBSE to repair sequential bugs and the existing work on healing and fixing concurrency bugs.

Sequential Bug Repair. Several approaches to sequential program repair have been proposed in the literature. For example, Arcuri and Yao as well as Wilkerson and Tauritz use co-evolutionary competition between programs with bugs (or between test cases) [2, 3, 25]. Both of these approaches require formal specifications and use genetic programming to evolve fixes.

Alternatively GenProg is another approach to sequential program repair but requires no formal specifications [24]. Instead, GenProg uses test cases to demonstrate a bug and describe the desired functionality that must be preserved. To address the limitations of the previous approach GenProg introduces two innovative features that allow the repair of real bugs in real programs: (1) It assumes the bug is written correctly in another part of the program and (2) It determines the error path on which the bug occurs and target only those statements for repair.

Concurrency Bug Repair. For concurrent programs, there are several examples of related work. One example is the use of ConTest to heal data races [16,18]. Healing a program is not the same as repairing a program – *"The healing techniques based on influencing the scheduling do not guarantee that a detected problem will really be completely removed, but they can decrease the probability of its manifestation"* [18].

AFix [14] is a framework for fixing single-variable atomicity violations in C++ programs. This approach combines dynamic bug analysis, patch creation and merging and dynamic testing. One limitation of AFix is that it only considers bug ides that involve manipulating mutex locks. We can not compare AFix with ARC because AFix work only on C++ programs which ARC works only on Java programs. Our analysis of the bugs capable of being fixed by AFix and ARC indicates that ARC can be applied to a wider variety of bugs and bug combinations (i.e., programs with different kinds of concurrency bugs present) and ARC offers a wider variety of possible bug fixes. Others have evaluated AFix and reported that, *"Our evaluation of AFix on large real systems also shows that the AFix sometimes incurs the degraded performance and, worse, frequent deadlocks"* [19].

Finally, Axis [19] is another concurrency bug repair tool that uses a branch discrete control theory called supervision based on place invariants to fix any number of correlated atomicity violations with minimal harm to concurrency. The Axis approach does not appear to be able to fix deadlocks.

8 Conclusions and Future Work

In this paper we have introduced ARC, a framework to automatically repair deadlocks and data races in concurrent Java programs. The goal of ARC is not only to ensure that a concurrency bug is repair but also to maximize the performance of the program once the bug has been fixed. To achieve this goal ARC consists of two phases:

1. *Phase 1:* a bug repair phase that employs a genetic algorithm without crossover to mutate an incorrect program, searching for a variant of the original program that fixes the deadlocks and data races.
2. *Phase 2:* an optimization phase attempts to optimize the concurrent source code by removing any excess synchronization without sacrificing program correctness. Excess and unneeded synchronization may be introduced in Phase 1 and can negatively affect performance.

To evaluate ARC, we conducted experiments using a set of 8 programs from the IBM concurrency benchmark. ARC was able to fix the data races and deadlocks in all 6 of the fixable programs. Although ARC was successful with the set of programs from the IBM concurrency benchmark we still need to evaluate ARC's scalability on larger open source projects. To assist with scalability we plan to leverage some of the different heuristics for seeding noise and different optimizations supported by ConTest [15]. These optimizations will hopefully reduce the testing time required to evaluate variants of the original program that are produced during both of ARC's phases.

Finally, we plan to further investigate the mutation operators used to repair concurrency bugs in ARC. Through experimentation we plan to optimize the existing set of mutation operators to maximize their capabilities while removing unnecessary operators. We also plan to experiment with new mutation operators that will increase the set of possible bug fixes. Potential additions to our current set of mutation operators include splitting or merging synchronization blocks and adding synchronize blocks with locks not used elsewhere in the program. Furthermore, we would like to expand ARC's operators to deal with new anti-patterns [5, 10] and give ARC the ability to fix additional types of bugs.

Acknowledgment. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Mutation analysis. Tech. rep., GIT-ICS-79/08, Georgia Institute of Technology (1979)
2. Arcuri, A.: On the automation of fixing software bugs. In: Proc. of Int. Conf. on Soft. Eng. (ICSE 2008), pp. 1003–1006 (2008)
3. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: Proc. of IEEE Congress on Evolutionary Computation (CEC 2008), pp. 162–168 (2008)
4. Bradbury, J., Cordy, J., Dingel, J.: Mutation Operators for Concurrent Java (J2SE 5.0). In: Proc. of the Work. on Mutation Analysis (Mutation 2006), pp. 83–92 (2006)
5. Bradbury, J., Jalbert, K.: Defining a Catalog of Programming Anti-Patterns for Concurrent Java. In: Proc. of the Int. Work. on Software Patterns and Quality (SPAQu 2009), pp. 6–11 (2009)
6. Cordy, J., Halpern, C., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. In: Proc. of the Int. Conf. on Computer Languages, pp. 280–285 (1988)

7. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java program test generation. IBM Systems Journal 41(1), 111–125 (2002)
8. Eytani, Y., Tzoref, R., Ur, S.: Experience with a concurrency bugs benchmark. In: Proc. of Software Testing Benchmark Work (TESTBENCH 2008) (2008)
9. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD 2004) (2004)
10. Fiedor, J., Křena, B., Letko, Z., Vojnar, T.: A uniform classification of common concurrency errors. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2011, Part I. LNCS, vol. 6927, pp. 519–526. Springer, Heidelberg (2012)
11. Galletly, J.: An overview of genetic algorithms. Kybernetes 21(6), 26–30 (1992)
12. Harman, M.: Why the Virtual Nature of Software Makes it Ideal for Search Based Optimization. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 1–12. Springer, Heidelberg (2010)
13. Havelund, K., Stoller, S., Ur, S.: Benchmark and framework for encouraging research on multi-threaded testing tools. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD 2003), pp. 22–26 (2003)
14. Jin, G., et al.: Automated atomicity-violation fixing. In: Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI 2011), pp. 389–400 (2011)
15. Krena, B., Letko, Z., Vojnar, T., Ur, S.: A platform for search-based testing of concurrent software. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD 2010), pp. 48–58 (2010)
16. Krena, B., Letko, Z., Tzoref, R., Ur, S., Vojnar, T.: Healing Data Races On-The-Fly. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD 2007), pp. 54–64 (2007)
17. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proc. of Int. Conf. on Soft. Eng. (ICSE 2012), pp. 3–13 (2012)
18. Letko, Z., Vojnar, T., Krena, B.: AtomRace: Data Race and Atomicity Violation Detector and Healer. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD 2008) (2008)
19. Liu, P., Zhang, C.: Axis: automatically fixing atomicity violations through solving control constraints. In: Proc. of Int. Conf. on Soft. Eng. (ICSE 2012), pp. 299–309 (2012)
20. Long, B., Strooper, P., Wildman, L.: A method for verifying concurrent Java components based on an analysis of concurrency failures. Concurrency and Computation: Practice & Experience 19(3), 281–294 (2007)
21. Musuvathi, M., Qadeer, S., Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software. Tech. rep., Microsoft Research (2007)
22. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proc. of ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2007), pp. 327–338 (January 2007)
23. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue 3(7), 54–62 (2005)
24. Weimer, W., et al.: Automatically finding patches using genetic programming. In: Proc. of Int. Conf. on Soft. Eng. (ICSE 2009), pp. 364–374 (2009)
25. Wilkerson, J., Tauritz, D.: Coevolutionary automated software correction. In: Proc. of Genetic and Evolutionary Computation Conf. (GECCO 2010), pp. 1391–1392 (2010)

A Modular Approach to Model-Based Testing of Concurrent Programs

Richard Carver[1] and Yu Lei[2]

[1] Dept. of Computer Science, George Mason Univ., Fairfax, VA, 22030
rcarver@cs.gmu.edu
[2] Dept. of Computer Science and Eng., Univ. of Texas at Arlington, Arlington, TX, 76019
ylei@cse.uta.edu

Abstract. This paper presents a modular approach to testing concurrent programs that are modeled using labeled transition systems. Correctness is defined in terms of an implementation relation that is expected to hold between a model of the system and its implementation. The novelty of our approach is that the correctness of a concurrent software system is determined by testing the individual threads separately, without testing the system as a whole. We define a modular implementation relation on individual threads and show that modular relations can be tested separately in order to verify a (non-modular) implementation relation for the complete system. Empirical results indicate that our approach can significantly reduce the number of test sequences that are generated and executed during model-based testing.

Keywords: Concurrent Programming, Model-Based Testing.

1 Introduction

A concurrent program contains two or more threads that communicate and synchronize with each other to perform some task. One general approach to testing a concurrent program is to execute the program with carefully selected test sequences. Model-based testing uses abstract models for test selection. That is, an abstract model is used to specify the intended program behavior, and test-sequences selected from the model are used to test a concrete implementation.

Abstract models for concurrent programs are often expressed as, or can be translated into, a labeled transition system (LTS). An LTS models program behavior as a type of state machine. Each state in an LTS is an abstraction of a state in the program. Transitions are labeled with the program events performed during state transitions. Our objective is to use model-based testing to determine whether a desired implementation relation exists between an abstract LTS model M and a concrete implementation CP. An example of such a relation is that the sequences of events allowed by M are also allowed by CP. When the implementation relation holds, we say that M is implemented by CP.

In this paper, we present a modular approach to model-based testing for concurrent programs that use message passing for communication and synchronization.

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 85–96, 2013.

We assume the existence of an abstract model M containing two or more LTSs, and a concrete implementation CP with two or more concurrent threads. We define a new type of implementation relation, called a modular implementation relation, between a single implementation thread of implementation CP and the corresponding LTS(s) in model M. Modular implementation relations are verified by generating modular test sequences from M and testing each implementation thread in CP separately. If all the modular relations hold, then a non-modular relation between M and CP is guaranteed to be satisfied.

Our test generation technique makes one important assumption, which is that the sole source of non-deterministic behavior in the model and the implementation is the order in which LTSs/threads synchronize and communicate. Other sources of non-deterministic behavior, such as uninitialized variables, are assumed to be absent.

The remainder of this paper is organized as follows. In Section 2, we show how the intended behavior of a concurrent program is modeled during modular testing. Section 3 defines a modular implementation relation and describes how modular testing can be used to verify this relation. Section 4 presents a technique for generating modular test sequences. Section 5 reports the results of an empirical study on modular testing. Related work is described in Section 6, including a comparison between two other modular testing techniques and our technique. Section 7 provides concluding remarks and our plans for future work.

2 LTS Models

The modular testing technique presented in this paper is for concurrent programs that use message passing for communication and synchronization. The intended execution behavior of a message-passing program is modeled using an extended LTS model called an annotated LTS [1], which is described below.

2.1 Labeled Transition Systems

LTS models contain nodes representing the state of a program and labeled edges representing transitions from state to state. LTSs can be composed in parallel and they can be synchronized by performing *matching* send and receive events, where *e* represents a synchronous receive event that matches synchronous send event *e'*. Synchronizations involving matching events are considered to be hidden from external observers and are represented as special τ (pronounced "tau") events. We assume that LTSs are composed using the laws in CCS (Calculus of Communicating Systems) [2] and an interleaving concurrency model.

Formally, an LTS is a 4-tuple $<Q,E,R,q_0>$, where Q is a non-empty finite set of states, E is a set of transition labels, $R \subseteq Q \times E \times Q$ is the transition relation, and q_0 is a state in Q denoting the initial state. For message-passing programs, the labels in E encode send and receive events. An LTS may contain one or more *termination states*, which are states without outgoing transitions. The τ events in an LTS are called *internal* events. All other events are called *external* events. A transition labeled with an internal (external) event is referred to as an internal (external) transition.

2.2 Adding Annotations

The send and receive events in an LTS model are encoded by simple transition labels. Formats that have been developed for representing test sequences for implementations encode send and receive events with more complex event descriptors, such as the ID of the sending or receiving thread, the operation performed, and the destination or source port of the operation [3], [4], [5]. (A port *p* is a communication channel through which messages are sent using *p.send()* and received using *p.receive()*. Only one thread can receive messages from a given port.) Event descriptors are not included in LTS models, but they are needed to transform abstract sequences of the model into concrete sequences of the implementation, and generate modular test sequences.

Koppol et al. [1] extended the LTS model and the algebraic laws used in CCS to allow implementation event descriptors to be encoded in an LTS. Their extended LTS model is called an annotated labeled transition system (ALTS). We use the ALTS model in this paper.

A formal definition of the ALTS model is given in [1]. Informally, an ALTS is an LTS in which each transition is annotated with information about the associated synchronization event in the implementation. A transition annotation has the form (L_i,L_j,port,op,label), where L_i is the sender and L_j the receiver for an operation *op* performed on *port* and labeled *label*. For synchronous message passing, *op* is either *synch_send* or *synch_receive*, or a synchronization between a *synch_send* and a *synch_receive*, denoted as a *synchronous-synchronization*. For asynchronous message passing, *op* is either *asynch_send* or *asynch_receive*. We use $E_{annotated}$ to denote the set of transition annotations in an ALTS. A technique for generating the annotations in an ALTS is described in [6].

ALTs are composed using the laws in CCS, with extra rules about forming annotations for synchronizations. For example, consider a composition of ALTS B_1 and ALTS B_2, denoted by $(B_1 \mid B_2) \setminus \{msg\_in\}$. A synchronization between B_1 and B_2 on receive event *msg_in (B1,B2,in,synch_receive,msg_in)* and send event *msg_in'(B1,B2,in,synch_send,msg_in)* results in a τ event with annotation $(B_1,B_2,in,synchronous\text{-}synchronization,msg\_in)$. The annotation for the τ event denotes that B_1 was the sender and B_2 the receiver for a synchronous synchronization labeled *msg_in* that occurred on port *in*. The annotation for the τ event carries the annotation information from the events that were synchronized to create it.

3 Modular Testing

Modular test generation begins with an abstract model M comprised of a set of ALTSs $\{L_1, L_2, \ldots, L_m\}$, and a concrete implementation CP of M with concurrent threads $\{P_1, P_2, \ldots, P_n\}$. To simplify our presentation, we assume that the number *n* of threads equals the number *m* of ALTSs and that thread P_i is mapped to ALTS L_i. We also assume the alphabets of labels for P_i and L_i are intended to be the same.

3.1 Implementation Relation M $\leq_F$ CP

The correctness of an implementation CP can be defined in terms of an implementation relation that is required to hold between CP and the ALTS model M = <Q,E,R,q_0> of CP. The set of all possible sequences that can be written using the labels in set $E_{annotated}$ of model M is denoted by $E_{annotated}$*.

Definition 1: A sequence *s* in $E_{annotated}$* is *feasible for model M* if *s* is a sequence of events along some path through M, starting at the start state of M; otherwise, *s* is *infeasible* for M.

Definition 2: A sequence *s* in $E_{annotated}$* is *feasible for implementation CP* if an execution of CP can exercise sequence *s*.

A *non-modular* implementation relation that is often used for test generation is denoted by M $\leq_F$ CP.

Definition 3: M $\leq_F$ CP $\equiv_{def}$ for any sequence *s* in $E_{annotated}$*: *s* is feasible for M $\Rightarrow$ *s* is feasible for CP.

Relation M $\leq_F$ CP requires each feasible sequence *s* of model M to be feasible for implementation CP. However, CP may have feasible sequences that are not feasible for M. This relation indicates perhaps that M is incomplete and thus is *extended* by CP, i.e., CP adds behavior that is not in M, but all the behaviors of M are still allowed by CP [7].

3.2 A Modular Implementation Relation for $\leq_F$

The implementation relation in Definition 3 is for the full model M and its implementation CP. In this section, we define an implementation relation for an individual thread P_i in CP and the ALTS L_i = <Q_i,E_i,R_i,q_i> in M to which P_i is mapped.

Definition 4: A *local* sequence with respect to ALTS L_i is a sequence $s_{Li} \in E_{annotated}$* such that all of the send and receive events in s_{Li} have L_i as the sender or receiver, respectively.

A sequence that is local with respect to ALTS L_i may or may not be allowed by L_i.

Definition 5: Let s_{Li} be a sequence that is local with respect to L_i. Local sequence s_{Li} is *feasible for* L_i if s_{Li} is a sequence of events along some path through L_i, starting at the start state of L_i; otherwise, s_{Li} is *infeasible* for L_i.

A feasible local sequence of L_i may not actually be allowed to occur when the constraints imposed on L_i by L_i's environment in M are considered. For example, L_i may allow two messages to be received in either order; while L_i's environment may require the first message to be received before the second message can be sent.

Definition 6: For feasible sequence *s* of M, the *projection of s onto* L_i is the (feasible) local sequence s_{Li} that is obtained by removing from *s* all of the send events for which L_i is not the sender and all of the receive events for which L_i is not the receiver.

- If *e* is an *asynch_send (asynch_receive)* event in *s* that is executed by L_i, then *e* is an *asynch_send* (*asynch_receive*) event in s_{Li}.
- If *e* is a *synchronous-synchronization* event in *s*, then *e* is a *synch_send* (*synch_receive*) event in s_{Li} if L_i executed the *synch_send* (*synch_receive*) event synchronized at *e*.

Definition 7: A feasible local sequence s_{Li} of L_i is *constrained with respect to model M* if s_{Li} is the projection onto L_i of some feasible sequence of M. The set of constrained sequences of L_i with respect to model M is denoted *Constrained-Sequences(L_i,M)*, or just *Constrained-Sequences(L_i)* when M is understood.

Definition 7 shows that Constrained-Sequences(L_i) is not determined by analyzing L_i and ignoring the other ALTSs in M. To the contrary, a sequence in Constrained-Sequences(L_i) must be a projection of some feasible sequence of M. Thus, Constrained-Sequences(L_i) captures the constraints imposed on L_i by the other ALTSs.

When a feasible sequence *s* of M is projected to obtain a constrained local sequence s_{Li} of L_i, the annotations on the events in *s* are retained by the events in s_{Li}. These annotations specify the interactions that occur between L_i and its environment when the events in s_{Li} are exercised. If L_i exercises a receive (send) event then the environment exercises a matching send (receive) event. An environment that interacts as specified by the annotations in s_{Li} is referred to as a *conforming* environment of s_{Li}.

Definition 8: A feasible local sequence s_{Li} in *Constrained-Sequences(L_i,M)* is feasible for implementation thread P_i if P_i can exercise sequence s_{Li} when P_i is executed with a conforming environment of s_{Li}.

A procedure for checking the feasibility of a constrained local test sequence for an implementation thread is given in Section 3.3.

Theorem 1: Let *s* be a feasible sequence of M and s_{Li} be the projection of *s* onto ALTS L_i, $1 \le i \le n$. Then constrained local sequence s_{Li} is feasible for thread P_i, $1 \le i \le n$, iff sequence *s* is feasible for CP.

Proof: The proof is omitted for the sake of brevity, but can be found in [8].

Based on Theorem 1, we can test each thread separately with the *constrained* local sequences of its corresponding ALTS instead of testing all the threads together with all of the feasible sequences of M. Building on this, we define a modular implementation relation for an ALTS L_i and the thread P_i to which it is mapped. This relation mirrors the relation in Definition 3:

Definition 9: $L_i \le_F P_i \equiv_{def}$ for any sequence s_{Li} in Constrained-Sequences(L_i): s_{Li} is feasible for P_i.

Modular implementation relation $L_i \le_F P_i$ is used in the following theorem, which is the basis for modular testing:

Theorem 2: $L_i \le_F P_i$, $1 \le i \le n$, iff $M \le_F CP$.

Proof: The if-part is obvious – Theorem 1 says that if a feasible sequence *s* of M is feasible for CP, then the constrained local sequences obtained by projecting *s* onto L_i, $1 \le i \le n$, are feasible for the individual threads of CP. It follows directly

from this that if all the feasible sequences of M are feasible for CP, then all of the constrained local sequences of L_i, $1 \le i \le n$, are also feasible for the individual threads of CP.

For the only-if part, assume relation $L_i \le_F P_i$, $1 \le i \le n$, holds but relation $M \le_F CP$ does not. Then there is an event e that is one of the (possibly many) events that can be the first event in some feasible sequence s of M that is not feasible for CP. (An event e is one of the first infeasible events in s if no event in s that happened before [9] e is infeasible. The possible first events are executed concurrently.) Assume that e is executed by ALTS L_j and let s_{Lj} be the projection of s onto L_j. Sequence s_{Lj} is a local sequence of L_j that is not feasible for P_i due to e, but we are assuming $L_j \le_F P_j$, which is a contradiction.

According to Theorem 2, the implementation relations between the individual threads in CP and the ALTSs in M can be verified separately in order to verify the implementation relation between M and CP. Testing each pair (L_i, P_i) separately is more efficient in cases where the local sequences of an ALTS L_i are repeated many times, perhaps even an exponential number of times, in the feasible sequences of M.

3.3 A Modular Testing Procedure for $M \le_F CP$

Modular testing is performed using the following procedure:

Procedure Test$_{\le F}$: For each mapped pair (L_i, P_i), $1 \le i \le n$:

(a) Generate Constrained-Sequences(L_i) (see Section 4.2).
(b) For each local test sequence s_{Li} in Constrained-Sequences(L_i):
  (b1) Test P_i with sequence s_{Li} and assign a test verdict, which is either *pass* or *fail*. The assignment of verdicts is discussed below.
  (b2) If P_i *fails* with s_{Li}, a failure has been detected in CP and testing halts.

Note that the key step in the above procedure is deriving Constrained-Sequences(L_i) in step (a), which is described in Section 4. In step (b1), thread P_i is executed with a test driver. The driver behaves as a conforming environment by supplying the send and receive events that match the events executed by P_i in local sequence s_{Li}. That is, whenever sequence s_{Li} calls for P_i to execute a send (receive) event on port p, a receive (send) event on port p is executed by the driver. The implementation information that is needed for mapping the abstract events in s_{Li} to concrete events of P_i is provided by the transition annotations in L_i, as described in Section 2. Note that the execution of thread P_i interacting with a test driver will be deterministic.

The test verdict in (b1) is assigned as follows:

`if` (P_i executes an event that is not in the alphabet E_i of L_i
  or local sequence s_{Li} is infeasible for P_i)
`then` the test *fails* `else` the test *passes*.

If procedure Test$_{\le F}$ is performed and all the tests in Constrained-Sequences(L_i) are passed for each pair (L_i, P_i), $1 \le i \le n$, then $L_i \le_F P_i$, and by Theorem 1, $M \le_F CP$.

4 Modular Test Generation Using Thread Interaction Models

In this section, we show how to build an ALTS model called a thread interaction model (TIM). The thread interaction model for Li, denoted TIM_{Li}, models L_i's interactions with the other ALTSs in M. Constrained-Sequences(L_i) is generated by traversing TIM_{Li}.

4.1 Using Reachability Analysis to Generate Thread Interaction Models

For each ALTS L_i in model M, we use equivalence-based reductions to build a thread interaction model TIM_{Li} that models L_i's interactions with the other ALTSs in M. The steps for building TIM_{Li} are as follows:

Step 1: Based on ALTS L_i, classify the transitions in M as observable or hidden.

For asynchronous message passing, the observable transitions are the send and receive transitions executed by L_i. For synchronous message passing, the observable transitions are the transitions that involve a synchronization in which L_i is the sender or the receiver. Other transitions are considered to be hidden. Thus, the observable transitions in M all involve interactions with L_i.

Step 2: Minimize M modulo observational equivalence [2], [10] creating ALTS model M_{Li}, which captures L_i's behavior in M.

Step 3: Minimize M_{Li} modulo weak-trace equivalence [10] creating ALTS TIM_{Li}.

When a minimization is performed in Step 2 or 3, the minimization is based on the *annotations* in the ALTSs. Recall that all the τ events will have the same label "τ" but different annotations. This allows τ events to be treated as different (observable) events during minimization. Thus, the annotation information about the events and the ALTSs that synchronize with L_i is retained in TIM_{Li}. This ensures that TIM_{Li} models all of L_i's interactions with other threads, and that TIM_{Li} contains the implementation information (in the form of annotations) that is necessary for generating concrete test sequences for implementation thread P_i.

The reduced thread interaction model TIM_{Li} produced in Step 3 represents the feasible sequences of interactions between L_i and the rest of the system. By definition of weak-trace equivalence, TIM_{Li} contains no hidden transitions and no redundant sequences of observable transitions. Algorithms for Step 2 run in time $O(n^3)$, where *n* is the number of states in the model. Algorithms for Step 3 have worst case running times that are exponential in the number of states, but the model minimized in Step 3 is a model of a single thread, which is typically much smaller than a global model. Thus, Step 2 should dominate the execution time.

4.2 Generating Test Sequences from Thread Interaction Models

Test sequences are derived by traversing TIM_{Li} and generating all the feasible sequences. For a cyclic model such as $TIM_{Li} = a.TIM_{Li}$, an exhaustive test suite would have infinitely many test sequences, each test sequence *a*, *a.a*, *a.a.a*, …, having a finite but arbitrarily long number of events. One approach for dealing with a cyclic

model M is to select a finite subset of M's test sequences, and ensure that cycles are iterated a finite number of times. Another approach is to redesign M as a model M' that is incomplete but that has an acyclic state space, and generate an exhaustive test suite from M'. The feasible sequences of acyclic model M' form a subset of the feasible sequences of the cyclic model M. In both approaches, a finite set of (finite-length) test sequences is generated; however, the sequences may fail to detect some errors.

If the TIM_{Li} generated by the process in Section 4.1 is *acyclic*, then it can be traversed using a simple depth-first search (DFS) algorithm. Whenever a termination state of TIM_{Li} is reached, the DFS backs up, and one or more test sequences are collected from the search stack. The collected test sequences include one complete sequence of TIM_{Li}, i.e., a sequence beginning at the start state of TIM_{Li} and ending in a termination state, and all of the non-null proper prefixes of this complete sequence. If TIM_{Li} is *cyclic*, then some test selection method must be used when TIM_{Li} is traversed to select a subset of the feasible test sequences of TIM_{Li}. Test selection may be based on, e.g., guidance from the person doing the testing [11-13], or coverage criteria [14].

Modular sequences are generated for ALTS L_i in M using the following procedure:

Procedure Generate_Sequences(L_i):
For each ALTS L_i in model M:

(G1) Apply Steps 1 through 3 in Section 4.1 to create thread interaction model TIM_{Li}
(G2) Traverse TIM_{Li} as described above to generate test sequences for L_i.

Whether procedure *Generate_Sequences(L_i)* generates Constrained-Sequences(L_i) depends on whether TIM_{Li} is acyclic.

Theorem 3: If TIM_{Li} is *acyclic*, then procedure *Generate_Sequences(L_i)* generates Constrained-Sequences(L_i) for each ALTS L_i, $1 \le i \le n$, in model M.

Proof: In procedure *Generate_Sequences(L_i)*, minimization modulo weak-trace equivalence in step (G1) is based on the annotations of the external transitions instead of the transition labels. This prevents any (non-redundant) sequences of L_i's transitions from being lost during the minimization. By definition of weak-trace equivalence, the traces of TIM_{Li} generated by the DFS procedure in step (G2) are precisely Constrained-Sequences(L_i).

We point out that an obvious optimization of the DFS search procedure is to avoid the generation of any sequence that is a prefix of a sequence that has already been generated. For example, if test sequence *a.b.c* is generated, it is not necessary to generate prefix sequences *a.b* and *a*. The reason being that if sequence *a.b.c* is feasible for the implementation thread, then sequences *a.b* and *a* must also be feasible. This optimization is easily performed during DFS by generating only complete sequences at backup points.

If TIM_{Li} is cyclic, then procedure *Generate_Sequences(L_i)* must use some test selection method to select a subset of the feasible sequences of TIM_{Li}. Thus, *Generate_Sequences(L_i)* will not generate Constrained-Sequences(L_i) or even all of the complete sequences of L_i. In this case, modular testing cannot be used to verify relation $M \le_F CP$; however, the generated sequences will be *sound*, i.e., only incorrect implementations will fail the tests [15].

5 Empirical Study

We conducted an empirical study in which modular test sequences were generated from thread interaction models. Abstract models and Java implementations were built for: *DP*: a deadlock-free solution to the dining philosophers problem with philosophers sharing forks and each philosopher eating once [16]; *DME-3*: a solution to the distributed mutual exclusion problem with three processes and three threads per process [17]; and *TDME*: a token-based solution to the distributed mutual exclusion problem [18] with 3 user processes and 1 controller process. All the models and implementations were acyclic. The DP and DME models were written in Lotos [19].

Our objective here was to study the effectiveness of modular tests for detecting violations of the implementation relations and to compare the number of test sequences generated by modular testing to the number of sequences generated by other approaches. The results show a range of results for reducing test set sizes, from a large reduction to no reduction. In this study, we leveraged several LTS reduction tools, and inherited their limitations, but we did not evaluate their scalability. That has been done by others [10], in many cases on real life, industrial systems.

Table 1 summarizes the results of test sequence generation.

Table 1. Results of modular test generation

	Global states/trans	TO-seqs.	PO-seqs.	Max TIM states/trans	Mod-seqs.
TDME	192 / 348	67,894	30	68 / 90	33
DP-3	76 / 126	238	6	8 / 8	9
DP-4	322 / 712	94,526	14	8 / 8	12
DP-5	1364 / 3770	108,549,484	30	8 / 8	15
DP-6	5778 / 19164	217,113,360,382	62	8 / 8	18
DME-3	367,733/1,403,821	> 3.5 trillion	4032	71 / 117	315

Global ALTSs were generated using standard interleaving semantics and then minimized modulo strong equivalence in order to remove redundant sequences. The resulting global ALTSs contained no internal events. The sizes of the global models are shown in column 1 of Table 1.

Each Lotos specification model was compiled into its individual ALTS components. The longest time for this step occurred while translating the Lotos DME-3 model into its 9 component ALTSs, which took a total of 7 minutes and 40 seconds on a 1.3GHx processor with 32 GB of RAM.

Non-modular test sequences were generated using two different methods. The first method reports the number of unique, *totally-ordered*, non-modular sequences generated from the ALTS models (column 2 of Table 1). For model DME-3, this procedure was unable to finish. Thus, we report our partial results as lower bounds on the number of sequences.

The second method reports the number of non-modular, partially-ordered sequences generated by the reachability testing algorithm in [5] (column 3 of Table 1).

The number of partially-ordered sequences was usually considerably smaller than the number of totally-ordered sequences. The partially-ordered sequences can be used to verify relation $M \leq_F CP$.

Thread interaction models were generated by following the step-by-step procedure in Section 4.1 using the CADP toolset [20-21] and the ALTS reduction tool in [1]. Generating modular tests from the thread interaction models using the optimized DFS procedure described in Section 4.2 and executing the tests against the implementations took only a few seconds.

Table 1 shows that the number of modular test sequences (column 5) was always significantly less than the number of totally-ordered sequences generated from the global models; and was significantly less than the number of partially-ordered sequences generated from three out of six of the global models. The number of states and transitions in the largest thread interaction model (TIM) is reported in column 4.The time for generating thread interaction models took 24 to 30 seconds.

The number of modular test sequences generated for model DP-P with P philosophers and P forks is always 3P. By way of comparison, a complete DP model has $2^P - 2$ partially-ordered sequences and considerably more totally-ordered sequences.

For the TDME model, most of the interactions are between the user processes and the controller process, the result being that the number of modular sequences of the controller process is the same as the number of unique, partially-ordered, non-modular sequences in the model. Since each of the three user processes has a single modular test sequence, the total number of modular tests sequences is 3 more than the number of partially-ordered, non-modular sequences.

For the DME-3 model, a total of 315 modular test sequences were generated for the nine implementation units. The global DME-3 model has 4,032 partially-ordered non-modular sequences and over 3.5 trillion totally-ordered, non-modular sequences. We measured the adequacy of the modular test-sequences generated for DME-3 by using mutation testing. Each mutant for DME-3 introduced a single change that was intended to simulate a programming error. Mutants were automatically generated by the Java-based mutation tool μJava [22]. Some of the mutants created were functionally equivalent to the original program. These mutants were identified and deleted, which left 190 mutants. We then applied modular testing to the nine threads in DME-3. A mutant was considered to be *killed* if a modular test sequence failed when it was executed against the thread that contained the mutant. Each of the 190 DME mutants was killed by the modular tests.

Finally, we discuss the threats to the validity of our case study. The main threat to external validity is the degree to which the subject programs are representative of true practice. The subject programs are small in terms of lines of code, but they represent complex, classical synchronization patterns and they illustrate well that the reduction in test sequences achieved by modular testing will vary from none at all to a significant amount. We plan to conduct experiments on more programs as an effort to reduce this threat. The main threat to internal validity is the possibility that errors were made in counting the test sequences. The partially-ordered sequences were counted using the reachability testing tool in [5]. The totally-ordered sequences were counted using a depth-first search algorithm, whose implementation was carefully tested.

6 Related Work

In this section, we briefly review existing work on modular, model-based testing of concurrent systems. We first note that there has been work in the area of compositional model checking. The basic idea is to verify the behavior of each module in isolation and infer global correctness properties of the whole system from the results of verifying individual modules. This typically requires the user to manually provide an assumption about the interaction between the module being checked and the rest of the system. Our modular testing technique automatically builds a thread interaction model using equivalence-based reductions of the whole system. Also, our test generation technique does not perform any model-checking. Model checking is complementary to our work — the abstract model can be verified using model checking before the model is used to generate modular tests for testing the implementation.

Several compositional conformance testing techniques have been developed [23], [24]. These techniques require each individual thread model to be *input enabled*, which means that each state must specify a response for every possible modeled input. One problem with this approach is that some inputs may be impossible in certain states, and it is not clear what response should be specified for an impossible input. Also, identifying impossible inputs manually is difficult when the possible inputs depend on complex interactions among multiple components. Another problem occurs when some inputs are available, i.e., messages have been sent, and their availability is not an error, but receiving and responding to these inputs is not allowed in a certain state. It is not clear how to specify in an input enabled model that certain available inputs are not allowed to be received or responded to. Our modular testing technique does not require LTS models to be input enabled, nor does it require implementation threads to have all inputs enabled in all states.

7 Conclusion

In this paper, we presented a modular approach to testing concurrent systems that are modeled as annotated labeled transition systems. This approach can be used to show that the non-modular sequences allowed by the complete model M are allowed by the complete implementation CP, and this can be done by testing the individual implementation threads separately, without testing the implementation as a whole. We plan to continue our work by developing a modular testing technique that can be used to check whether M and CP allow the same sequences.

References

1. Koppol, P.V., Carver, R.H., Tai, K.C.: Incremental Integration Testing of concurrent Programs. IEEE Transactions on Software Engineering 28(6) (2002)
2. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
3. Tai, K.C.: On testing concurrent programs. In: Proc. COMPSAC, vol. 85, pp. 310–317 (1985)

4. Tai, K.C., Carver, R.H.: Testing of distributed programs. In: Zoyama, A. (ed.) Handbook of Parallel and Dist. Computing, ch. 33, pp. 955–978. McGraw Hill (1996)
5. Lei, Y., Carver, R.H.: Reachability testing of concurrent programs. IEEE Transactions on Software Engineering 32(6), 382–403 (2006)
6. Chen, J., Carver, R.: Selecting and Mapping Test Sequences from Formal Specifications of Concurrent Programs. In: Proc. of the High-Assurance Systems Eng., pp. 112–119 (1996)
7. Brinksma, E.: A Theory for the Derivation of Tests. In: Aggarwal, S., Sabnani, K. (eds.) Protocol Specification, Testing and Verification, VIII, pp. 63–74 (1988)
8. Carver, R., Lei, Y.: A Modular Approach to Model-Based Testing of Concurrent Programs. Technical report GMU-CS-TR-2013-5, Dept. of Computer Science, George Mason University (2013)
9. Lamport, L.: Time, Clocks, and the Ordering of Events in a Dist. System. Comm. ACM, 558–565 (1978)
10. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A Semantics Tool for the Verification of Concurrent Systems. ACM Tran. Programming Languages and Systems 15(1), 36–72 (1993)
11. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. International Standard IS-9646. ISO, Geneve (1991)
12. Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J.: Test Selection, Trace Distance and Heuristics. In: Proc. IFIP 14th Int. Conference on Testing Communicating Systems - TestCom, pp. 267–282 (2002)
13. Tretmans, J., Brinksma, E.: TorX: Automated Model-Based Testing. In: Proc. First European Conference on Model-Driven Software Engineering, pp. 31–43 (2003)
14. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge Univ. Press (2008)
15. Tretmans, J.: Testing Concurrent Systems: A Formal Approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 46–65. Springer, Heidelberg (1999)
16. Carver, R., Tai, K.C.: Modern Multithreading: Wiley (2006), `http://www.cs.gmu.edu/~rcarver/ModernMultithreading/`
17. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM 24(1), 9–17 (1981)
18. Turner, K.J.: Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL. John Wiley & Sons, Inc., New York (1993)
19. Suzuki, I., Kasami, T.: A distributed mutual exclusion algorithm. ACM Transactions on Computer Systems 3(4), 344–349 (1985)
20. Fernandez, J., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP: A Protocol Validation and Verification Toolbox. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 437–440. Springer, Heidelberg (1996)
21. Lang, F.: Compositional Verification using SVL Scripts. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 465–469. Springer, Heidelberg (2002)
22. Ma, Y.-S., Offutt, J., Kwon, Y.-R.: μJava: An Automated Class Mutation System. Journal of Soft. Testing, Verif. and Reliability 15(2), 97–133 (2005), `http://ise.gmu.edu/~ofut/mujava/`
23. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional Testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
24. Gotzhein, R., Khendek, F.: Compositional Testing of Communication Systems. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 227–244. Springer, Heidelberg (2006)

A Dynamic Approach to Isolating Erroneous Event Patterns in Concurrent Program Executions

Jing Xu[1], Yu Lei[1], Richard Carver[2], and David Kung[1]

[1] Dept. of Computer Science and Engineering, University of Texas at Arlington, USA
[2] Dept. of Computer Science, George Mason University, USA

Abstract. Concurrency bugs are hard to find due to the nondeterministic behavior of concurrent programs. In this paper, we present an algorithm for isolating erroneous event patterns in concurrent program executions. Failed executions are characterized as a sequence of switch points, which capture the interleaving of read and write events on shared variables. The algorithm inputs the sequence of a failed execution, and outputs erroneous event patterns. We implemented our algorithm and conducted an experimental evaluation on several Java benchmark programs. The results of our evaluation show that our approach can effectively and efficiently identify erroneous event patterns in failed executions.

Keywords: Concurrency, Fault Localization, Debugging.

1 Introduction

As concurrent programs become widespread, it is important to have effective and efficient techniques and tools for testing and debugging concurrent programs. A survey from Microsoft [17] reveals that nearly two-thirds of Microsoft programmers have to deal with concurrency issues and over half of the programmers detect, debug, and fix concurrency faults on at least a monthly basis. Furthermore, over 60% of these faults take several days to fix. Failures caused by concurrency faults can have potentially devastating consequences. For example, in 2003, a blackout in the northeastern U.S. left tens of millions of people without electricity, due to a race condition in the power plant monitoring software [18].

Much work has been reported on detecting two types of concurrency fault. One is atomicity violation (also referred to as serializability violation), which occurs when a code block that is expected to be executed atomically is executed non-atomically. The other is order violation, which occurs when code blocks are executed in an incorrect order. Recent work uses dynamic pattern analysis [2, 3, 6, 22] to recognize patterns of events that may be associated with these faults. Some pattern analysis techniques try to extract a set of benign patterns from passed executions and then identify violations of these benign patterns in failed executions [3, 6, 22]. However, some patterns may appear in both passed and failed executions [2]. These patterns will be classified as benign, which prevents them from being identified in failed executions.

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 97–109, 2013.

We propose a dynamic approach to identifying the erroneous patterns in a failed execution. Our approach is independent from the underlying synchronization mechanisms such as shared variables, semaphores, and monitors. The main idea of our approach is described as follows. We record the trace of a failed execution as a sequence of events. A systematic search strategy is used to find the erroneous switch point that causes the execution to fail. The novelty of our approach is the use of a notion called least concurrency mode to determine the correctness of a switch point. In the least concurrency mode, each thread executes until it cannot proceed further, i.e., it either blocks or finishes. The motivation is to minimize the number of times a thread being interrupted by another thread and thus reduce the chance of failure due to concurrency. To determine the correctness of a switch point s, we perform a number of test executions in which we first replay all the events up to and including s, which allows s to be reproduced, and then let the program proceed in the least concurrency mode. If one of these executions passes, s is likely to be a benign switch point. Otherwise, s is likely to be an erroneous switch point. After identifying the erroneous switch point, our technique tries to find erroneous event patterns related to this erroneous switch point, which can help users localize the faults.

This fault localization technique has been implemented in a tool called *Huatuo*, which was used to perform an empirical study on 13 benchmark Java programs. The results of the study show that our technique can effectively and efficiently localize the faults in 12 of the 13 programs.

2 Preliminaries

Let *s* be a totally-ordered sequence of read and write events in a failed execution of concurrent program CP with switch points P_1, P_2, … P_n, $n \geq 1$. Switch point P_i in *s* is *erroneous* if CP enters an incorrect internal state, called an *error* state, after P_i is executed, but before P_{i+1} is executed, and this error state is propagated to cause CP to output an incorrect result (a *failure*) [21].

After a switch point CP may enter an error state, which however may not propagate to the result. This case is possible because CP may recover from an error state, or the result is "coincidentally correct" and CP does not fail.

An atomicity violation occurs when a sequence of read and write events is executed in a way that cannot be serialized and this sequence violates the programmer's intention of atomicity. Figure 1 shows the five possible patterns of unserializable interleavings.

If two threads each access a shared variable, and at least one of the two accesses is a write access, then these two accesses comprise a conflicting interleaving pattern. An order violation occurs when a conflicting interleaving pattern is executed and this pattern violates the programmer's intended ordering.

An event pattern is *erroneous* if the appearance of this pattern causes the execution to enter an error state. Patterns that are not erroneous are *benign*.

	Interleaving	Description
1	T1:R R T2: W	Two reads by T1 were expected to have the same value.
2	T1:W R T2: W	The read by T1 was expected to read the value written byT1.
3	T1:W W T2 R	A temporary result written by T1 was not expected to be read by T2.
4	T1:R W T2 W	The value written by T2 was unexpectedly overwritten by T1.
5	T1:W W T2: W	The value written by T2 was unexpectedly overwritten by T1

Fig. 1. Unserializable Interleaving Patterns [5]

3 A Motivating Example

As a motivating example, we consider a Java class *Account* in Figure 2. This class is from the *ConTest* benchmark programs [19]. Class *Account* has two fields *amount* and *name*, and three methods *deposit*, *withdraw* and *transfer*. Method *deposit* adds a given amount of money, method *withdraw* withdraws a given amount of money and method *transfer* transfers a given amount of money from one account to another.

```
public class Account {
  double amount;
  String name;
  public Account(String nm, double amnt ) {
        amount = amnt;
        name = nm;
  }
  synchronized void deposit(double money){
        amount += money;
  }
  synchronized void withdraw(double money){
        amount -= money;
  }
  synchronized void transfer(Account ac, double mn){
        amount -= mn;
        ac.amount += mn;
  }
}
```

Fig. 2. An example (faulty) program

Figure 3 shows a failed execution in which two accounts, account1 and account2, are accessed concurrently by two threads. Thread 1 (or thread 2) initializes account1

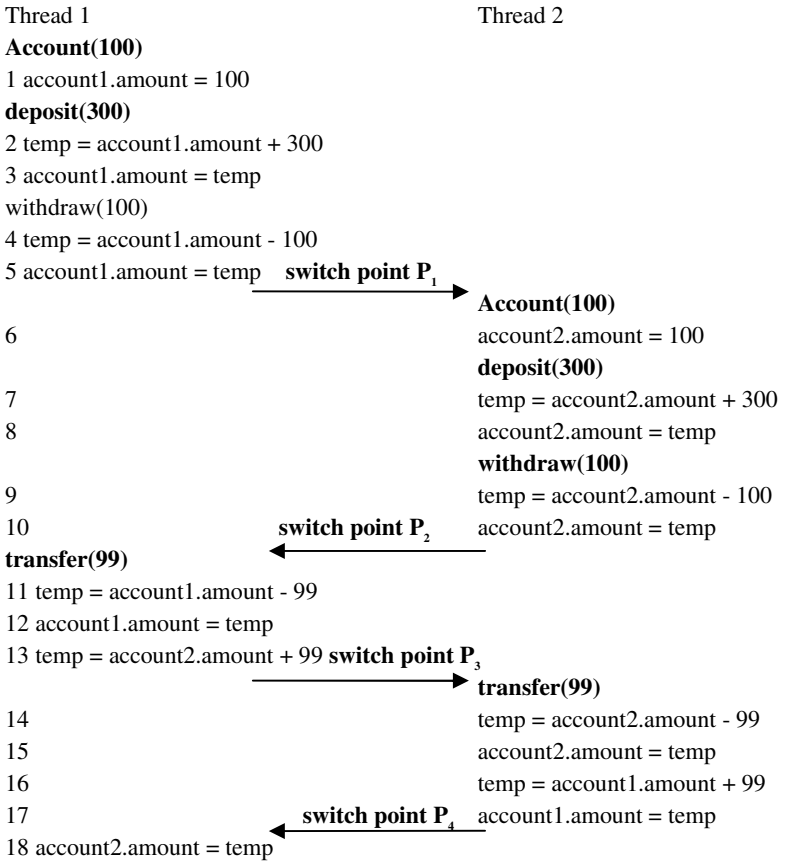


Fig. 3. A failed execution with class *Account*

(or account2) with 100, deposits 300, withdraws 100 and then transfers 99 to the other account. However, the final balance of *account2* is 399, instead of 300. This is because method *transfer* directly accesses *ac.amount,* where *ac* is an *Account* object passed to method *transfer* as an argument. In our example scenario, when Thread 1 calls method *transfer*, it only acquires the lock for *account1*. So the lock for *account2* can still be acquired by Thread 2, which can access and modify *account2* concurrently with thread 1.

Below we illustrate how our approach is used to identify the erroneous patterns in the example execution shown in Figure 3. Our technique has two phases. In the first phase, we identify the erroneous switch point. In the second phase, we identify the erroneous event patterns.

Switch points are checked in the reverse order of their occurrence in the failed execution. Thus, the first switch point checked is P_4. Step 1 of the controlled execution for P_4 replays the execution up to and including the execution of statement 18 in Thread 1 (We will explain how to obtain the replay portion of a failed execution

in the Section IV.). This ensures that switch point P_4 appears in the new execution. In step 2 of the execution, a "least concurrency" policy is enforced. Since there are no more statements to execute, the least concurrency part of the controlled execution does not exercise any events. The resulting execution fails, allowing us to conclude that switch point P_4 or one of the switch points that precede P_4 is erroneous.

Next we generate a controlled execution to check switch point P_3 of the failed execution. Step 1 of the controlled execution replays the events up to and including the execution of statement 17 in Thread 2. This ensures that switch point P_3 appears in the new execution. Step 2 of the controlled execution enforces the "least concurrency" policy, which executes Thread 1 until it ends. The new execution fails, which allows us to conclude that switch point P_3 or one of the switch points that precede P_3 is erroneous.

Next we generate a controlled execution to check switch point P_2. Step 1 of the controlled execution replays the events up to and including the execution of statement 13 in Thread 1. This ensures that targeted switch point P_2 appears in the new execution. Step 2 of the controlled execution enforces the "least concurrency" policy, which can force Thread 1 to execute until it ends and then Thread 2 to execute until it ends. This controlled execution passes. We also can force Thread 2 to execute first in the least concurrency mode and the generated execution is failed. Since we find one passed execution for switch point P_2, switch point P_2 is benign, and we conclude that switch point P_3, which is the switch point immediately after benign switch point P_2, is the erroneous switch point.

Next, we try to detect erroneous event patterns related to erroneous switch point P_3. Our technique checks whether there are any unserializable interleaving patterns or conflicting interleaving patterns that are introduced by P_3. In Figure 3, statement 13 in Thread 1, statement 15 in Thread 2 and statement 18 in Thread 1 comprise an unserializable interleaving pattern of shared variable *account2.amount* introduced by switch point P_3. This implies that the value written by statement 15 in Thread 2 is unexpectedly overwritten by the write operation of statement 18 in Thread 1, i.e., the write operation of statement 15 in Thread 2 should not interrupt the execution of statements 13 and 18 in Thread 1. This pattern is output to the programmer to guide debugging.

Delta debugging [1] fails to point out P_3 is the erroneous switch point. Assume the alternative passed execution is generated with switch points 5,11,13,16, while those for the failed execution in Figure 3 are 5,10,13,17. Since both executions have a switch point P_3 at event 13 and delta debugging only detects the erroneous switch point from the difference between the schedules of two executions, delta debugging would point out that P_2 in the failed execution is the cause of failure, which however does not make the execution enter erroneous state. The fundamental reason is that delta debugging only identifies the switch point that differs between the passed and failed execution and that if reconciled, would flip the result of the executions. Such a switch point does not necessarily create an erroneous state.

4 Our Algorithm

In this section, we describe our algorithm for fault localization in detail. Figure 4 shows algorithm *LocalizeErroneousPatterns*. This algorithm takes as input a program P and a totally-ordered sequence F of read and write events exercised by a failed execution of P. The output is a set of erroneous event patterns that trigger the failure. *LocalizeErroneousPatterns* has two major phases: (1) identifying the erroneous switch point; and (2) identifying the erroneous event patterns.

LocalizeErroneousPatterns begins by identifying the switch points in sequence F. A prefix *of F* is generated for each of the identified $switchPoint_i$ (lines 2-4). For $switchPoint_i$, the prefix contains all the events up to and including the event that immediately precedes $switchPoint_{i+1}$. The prefix for $switchPoint_i$ is used to replay the portion of F that contains $switchPoint_i$.

In phase 1, *LocalizeErroneousPatterns* checks switch points in the reverse order as they appear in sequence F (line 5-13). This process stops when it finds that $switchPoint_i$ is benign (line 8) and records $switchPoint_{i+1}$ as the erroneous switch point (line 9).

Algorithm *LocalizeErroneousPatterns*:
Input: program P, a totally ordered sequence F from a failed execution of P.
Output: a set *erroneousPatterns* of erroneous event patterns

1. Let *switchPoints* be a sequence of switch points in their order in F;
2. for ($switchPoint_i$ in *switchPoints*){
3. create the prefix for $switchPoint_i$
4. }
5. for (int i = number of switch points; i >= 1; i --) {
6. for (int j = 0; j < number of shared variables; j ++) {
7. let P replay $prefixFile_i$ first and then execute in the least concurrency mode
8. if(the generated execution passes){
9. *record* $switchPoint_{i+1}$ as the erroneous switch point
10. break the outer for loop;
11. }
12. } // end inner for
13.} // end outer for
14. if(all the switch points are erroneous){
15. record $switchPoint_1$ as the erroneous switch point
16. }
17. *erroneousPatterns* = {patterns collected with the erroneous switch point }
18. return *erroneousPatterns*

Fig. 4. Algorithm LocalizeErroneousPatterns

A controlled execution is used to replay the prefix generated for a targeted switch point and then force the execution to proceed in the least concurrency mode (line 7). The reason why the least concurrency mode is used after replaying the prefix is as follows. Atomicity violation and order violation are both due to interleavings of concurrent shared variable accesses. Assume that switch point P_i is targeted, and that switch point P_{i+1} is the erroneous switch point. When the prefix for P_i is replayed, and the least concurrency mode is used to complete the execution, interleavings of shared variable accesses are minimized. This means that additional switch points, including erroneous switch point P_{i+1}, can be avoided after P_i is replayed, allowing the execution to pass. If the prevention of P_{i+1} from being executed allows the execution to pass, switch point P_{i+1} is identified as the erroneous switch point.

Multiple controlled executions are used to determine whether a targeted switch point is benign or not. The following heuristic is used to identify benign switch points: if at least one controlled execution passes for a targeted switch point (it means the execution with the targeted switch point can pass), then we conclude that this switch point is benign and that the other failed controlled executions for this switch point are due to the event patterns introduced during the least concurrency mode. Likewise, if all the controlled executions for a targeted switch point fail, then this switch point is identified to be the erroneous switch point. With this heuristic, we ignore the possibility that the target switch point is benign and all the failures are due to the least concurrency part. From the empirical study results in section 5, our algorithm works effectively under this heuristic. Based on this heuristic, *LocalizeErroneousPatterns* performs controlled executions until an execution passes, or a maximum number of executions, which equals the number of shared variables, have been performed (line 6). This limit is set to the number of shared variables because of the strategy we use to generate controlled executions, which is described as follows.

For each shared variable *s*, the last thread L that accessed *s* in the replay mode is allowed to execute first in the least concurrency mode and continue execution until it blocks or terminates. In this way, no accesses from other threads can interrupt the access of *s* by L and we can avoid any potential erroneous event patterns for *s* that may occur at the boundary between the replay part and the least concurrency part in the controlled execution. The threads that execute after L are randomly selected. The number of controlled executions required by this strategy is equal to the number of shared variables. Note that this strategy does not allow us to determine with certainty whether a targeted switch point is erroneous. However, the empirical study in Section 5 suggests that this strategy can be effective for many programs.

As we mentioned above, switch points are targeted in the reverse order of their appearance in F. If switch points were instead checked in the order as they appeared in F, we could not conclude that the first switch point P_i that makes all the controlled executions fail is the erroneous switch point in F. This is because some switch point P_j after P_i may allow the execution to recover, making a later switch point P_k the cause of the failure. When switch points are checked in the reverse order, we can conclude that the switch point P_{i+1} that follows the first benign switch point P_i is the switch point that caused the original execution to fail. This is because after P_{i+1} is introduced

into the executions all the controlled executions fail, which indicates that P_{i+1} is the switch point that causes the failure.

We point out that binary search cannot be used to find the erroneous switch point. During our detection, the executions generated for each switch point can be all failed or contain at least one passed execution, because the execution can recover from some erroneous states. Since binary search can only be applied when the elements are sorted, our search process cannot use binary search. For example, if we test all the switch points in a failed execution, we may get the following result: P P F F F P F F F. (P represents the switch point is benign and F represents the switch point is erroneous) It indicates that the error introduced by switch point 3 can be recovered by switch point 6. Applying binary search, we will identify switch point 3 as the erroneous switch point. But this error cannot be seen from the output.

After an erroneous switch point is identified, phase 2 of *LocalizeErroneousPatterns* identifies the erroneous event patterns related to the erroneous switch point. Let switch point P_i be the erroneous switch point. Assume that thread A executes between switch point P_{i-1} and P_i, and thread B executes between switch point P_i and P_{i+1}. To identify unserializable interleaving patterns, which are associated with atomicity violations, for each shared variable we select one event from each of the following three blocks of events: (1) the events executed by Thread A between switch point P_{i-1} and P_i; (2) the events executed by Thread B between switch point P_i and $P_{i+1;}$ and (3) the events executed by Thread A between switch point P_j and P_{j+1}, where P_j is the first switch point at which control switched back to Thread A after switch point P_i. The reason why the algorithm can detect unserializable interleaving patterns from these three blocks of events is because all the unserializable interleaving patterns share the following property. Referring to the five unserializable interleaving patterns in Figure 1, for each pattern, after the first two events are exercised, the execution enters an erroneous state. For example, for the pattern (T1:R)-(T2:W)-(T1:R), the two reads by T1 expect to read the same value, but after the execution of (T1:R)-(T2:W), the second read by T1 cannot read the same value as the first read. Thus, if a failed execution has an unserializable interleaving pattern, the erroneous switch point found by our algorithm is the switch point between the first and the second event in the pattern, and the third event will be executed by the same thread as the first event.

After the three blocks of events are identified, for each shared variable, the algorithm selects one event from each of the three blocks, which is the same scheme used by Falcon [5]. Preference is given to the selection of write events, since unserializable interleaving patterns require at lease one write event [5]. The algorithm then checks whether the selected events comprise an unserializable interleaving pattern. All the patterns for all the shared variables are output to the user to guide debugging. If we cannot find any unserializable interleaving patterns, we search for conflicting interleaving patterns, which are associated with order violations, with a similar process, but conflicting interleaving pattern only contains two events.

5 Empirical Study

Our fault localization algorithm has been implemented in a tool called *Huatuo*. As a proof-of-concept, we used *Huatuo* to conduct several empirical studies of our fault localization technique on a suite of faulty multithreaded Java programs. Our objective was to investigate the following two questions:

1. What is the most effective and efficient technique for controlling thread executions during the least concurrency mode?
2. How effective and efficient is algorithm *LocalizeErroneousPatterns* at finding the faults?

Since tools are not available for the techniques most closely related to ours, such as replay analysis [23], Falcon [5], and delta debugging [1], we are not able to compare experimental results from our technique with results from the other techniques. In section 6, we will compare our approach to these techniques.

We selected 13 programs for our empirical study, all of which are faulty concurrency programs that are used in [5]. Compared to the empirical study in [5], we removed the following programs: (1) Hedc, which has a bug hidden in the library code and we cannot instrument the library code; (2) Philo and Tsp, both of which never failed even though we executed them for four hours; and (3) TreeSet, which has the same bug with HashSet in the super class collection [9]. Note that for program BufWriter, the main() function calls Thread.sleep() to give its child threads time to finish. Our tool cannot deal with operation sleep(), so we modified BufWriter to use join() instead of sleep(). In order to conduct the empirical study, we rewrite these programs using the Modern Multithreading library. The failed executions were traced and recorded using the Modern Multithreading library. We manually inserted some assertions in the programs to determine whether a test execution fails or succeeds.

5.1 Study 1: Selecting the First Thread to Run in the Least Concurrency Mode

The goal of this empirical study was to answer Question (1). To answer Question (1), we implemented two techniques for controlling thread executions during the least concurrency mode. Technique 1 used the strategy described in Section 4 for controlling thread executions. This strategy makes a careful selection of the first thread to execute during controlled executions. Technique 2 was to perform five test executions for each switch point, with the first thread to execute randomly selected.

Table 1 shows the result of this study. The first column identifies the subject programs. The second column shows the size of each program in terms of lines of code (LOC). The third column shows the number of shared variables in each program. The shared variables were identified manually based on documentation and source code. The fourth column shows the number of threads for each program. The fifth column shows the total number of executions when technique 1 was used to search for the erroneous switch point. The sixth column shows whether technique 1 can

successfully identify the erroneous event patterns. Columns 7 and 8 show the results for technique 2 in the same format.

Table 1 shows that both techniques can successfully find the erroneous switch points and the erroneous event patterns, except for program RayTracer. This is because RayTracer contains a large number of threads and at some switch points the controlled execution can pass only if a specific thread is executed first in the least concurrency mode. Thus, the probability that the first thread chosen by technique 2 results in a passed execution is low. As we mentioned above, five controlled executions were performed at each switch point using technique 2. However, this was not enough for technique 2 to generate a passed execution. Technique 1 generated a passed execution for RayTracer, and overall, technique 1 only required half of the number of executions required by technique 2 for generating a passed execution for the subject programs.

Table 1. Comparison between two strategies for controlled executions

Program	LOC	# of shared variables	# of threads	Technique 1		Technique 2	
				Total # of executions	success	Total # of executions	success
Account	177	2	3	8	Yes	17	Yes
AirlineTickets	142	2	7	14	No	40	No
BubbleSort2	184	3	3	13	Yes	21	Yes
BufWriter	183	3	3	9	Yes	17	Yes
Lottery	154	2	3	11	Yes	23	Yes
MergeSort	375	3	4	10	Yes	16	Yes
Shop	226	11	3	10	Yes	22	Yes
Arraylist	5898	3	3	8	Yes	20	Yes
HashSet	7103	10	3	7	Yes	11	Yes
StringBuffer	1380	33	3	13	Yes	21	Yes
Vector	760	5	3	10	Yes	16	Yes
Cache4j	3976	2	2	3	Yes	6	Yes
RayTracer	2047	2	17	14	Yes	41	No

5.2 Study 2: Effectiveness and Efficiency

The results in Table 1 show that our technique correctly identified the erroneous event patterns for all the programs except program *AirlineTickets*. Program *AirlineTickets* fails even when the program executes serially and all of the passed executions need an extra common switch point that is missing in the failed executions. So all the controlled executions are failed, which make our algorithm to conclude that the first event after the first switch point is erroneous. But the real fault is that it needs a context switch at a specific point. Also, our algorithm identified a single erroneous event pattern for all the programs except for program *Bubblesort2*, for which two

patterns were identified, which means one or both of the patterns can help the user localize the faults. In general, the algorithm does not make assumptions about what synchronization mechanisms are used in the concurrent program and can localize the faults for 12 out of 13 programs except the one that cannot pass when it is executed serially.

The instrumentation for the replay and the least concurrency mode slows down the executions. We performed 6 executions on the original version and the instrument version of program *RayTracer* which was the largest program in our benchmark. The slowdown factors are 22.4, 20, 21.5, 25.2, 26.4, and 25.25. On average, our instrumented executions took 23.5x longer than non-instrumented executions. This is faster than the results in [3, 11, 24], which reported average slowdowns from 25x [3] to more than 200x [24]. Although the slowdown of Falcon [2] is 9.9x, which is faster than our tool, our systematic search technique needs fewer executions than the techniques based on training [2, 3].

6 Related Work

In [23], the authors tried to distinguish benign races from erroneous ones after they detect all the data races. They execute a program twice for a given data race — once for each of the two possible orders of conflicting memory operations. However, when testing the alternative order, there is no guarantee the alternative execution is feasible. As mentioned in section 4.2.1 in [23], the alternative execution is possible to follow a totally different data and control flow, making it impossible to execute the alternative order of conflicting memory operations. The authors classify this as a replay failure, and in their experiments, 29 benign data races were potentially harmful races. Using the least concurrency mode of execution, our technique can determine whether or not a data race is benign by only replaying the orders of conflicting memory operations that appear in the failed execution, without having to test the alternative orders.

The AVIO method [3] uses heuristics to automatically extract access interleaving invariants and detect violations of these invariants at run time. Defuse [22] uses training to learn definition-use invariants and considers violations of these invariants to be erroneous. Since both AVIO and Defuse are invariant-based approaches, they can only report erroneous patterns that only appear in failed executions. Our technique can identify erroneous patterns in the failed execution, even when these patterns also appear in passed executions. The reason why AVIO and Defuse may miss some patterns is because it is assumed that any execution that contains an erroneous pattern will fail. However, this is not always true. A pattern that triggered the failure in a failed execution can also appear in passed executions [5].

Falcon [5] monitors memory-access sequences among threads, detects data-access patterns associated with a program's pass/fail results, and ranks data-access patterns with regards to how suspicious they are. The main drawback of this technique is that highly suspicious patterns may not be the patterns that caused a failure; rather, they may be patterns that are resulted from the erroneous patterns. Our technique systematically tests the switch points in the failed execution one by one, and can pinpoint the actual erroneous pattern that triggers the failure.

By systematically narrowing down the difference between a failed thread schedule and a passed thread schedule, the Delta Debugging approach [1] can pinpoint the thread switch that differs between the two schedules and that if reconciled, would flip the result of the two schedules. As shown in section 3, such a thread switch may not actually produce an erroneous state. Our technique can find the switch point and the event pattern that actually produces an erroneous state which we believe is of more help for debugging.

7 Conclusion

In this paper, we presented an algorithm for identifying erroneous event patterns in concurrent executions. Failed executions are characterized as a sequence of events, which capture the interleaving of read and write events on shared variables. The algorithm inputs the sequence of switch points of a failed execution, and then uses controlled executions to distinguish erroneous switch points from benign switch points. The output of the algorithm is the erroneous event patterns. The event pattern can guide the user in locating the actual fault that triggered the failure. The algorithm is implemented in a tool called *Huatuo*. The results of our empirical study show that Huatuo can effectively and efficiently identify erroneous event patterns.

There are a number of venues to continue our work. First, we plan to conduct more experiments to evaluate the effectiveness of our approach. In particular, we want to conduct experiments on more complex real-life programs. Second, our approach currently deals with switch points, and a concurrent execution may consist of a large number of switch points. We will explore the idea of grouping switch points. Doing so will help reduce the number of switch points that have to be checked by our approach. Finally, we want to further develop our prototype tool and release it as an open-source tool.

References

1. Choi, J., Zeller, A.: Isolating Failure-Inducing Thread Schedules. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 210–220 (July 2002)
2. Park, S., Vuduc, R., Harrold, M.: A Unified Approach for Localizing non-deadlock Concurrency Bugs. In: 2012 IEEE Fifth International Conference on Proceedings of the Software Testing, Verification and Validation (ICST), pp. 51–60 (April 2012)
3. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 37–48 (December 2006)
4. Park, S., Lu, S., Zhou, Y.: CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In: Proceedings of ISSTA 2002 Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 25–36 (March 2009)
5. Park, S., Vuduc, R., Harrold, M.: Falcon: Fault Localization in Concurrent Programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 245–254 (2010)

6. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ACM/IEEE 30th International Conference on Proceedings of the ICSE 2008, pp. 231–240 (May 2008)
7. Lai, Z., Cheung, S., Chan, W.: Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 235–244 (2010)
8. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: Proceedings of the Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 334–345 (2006)
9. Sen, K.: Race Directed Random Testing of Concurrent Programs. In: Proceedings of PLDI 2008, pp. 11–21 (2008)
10. Carver, R.H., Lei, Y.: A Class Library for Implementing, Testing, and Debugging Concurrent Programs. Int. Journal on Software Tools for Tech. Transfer 12(1), 69–88 (2010)
11. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL, pp. 256–267 (2003)
12. Wang, L., Stoller, S.D.: Static analysis for programs with non-blocking synchronization. In: PPoPP (2005)
13. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. Trans. Comput. Syst. 15(4), 391–411 (1997)
14. Ronsse, M., Bosschere, K.D.: RecPlay: a fully integrated practical record/replay system. Trans. Comput. Syst. 17(2), 133–152 (1999)
15. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (June 2007)
16. Flanagan, C., Freund, S.N.: Type-based race detection for java. In: PLDI, pp. 219–232 (June 2000)
17. Godefroid, P., Nagappan, N.: Concurrency at Microsoft: An exploratory survey. In: Workshop on Exploiting Concurrency Efficiently and Correctly (2008)
18. Poulsen, K.: Tracking the blackout bug. SecurityFocus (February 2004), http://www.securityfocus.com/news/8412
19. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Concurr. Comput.: Pract. Exper. 19(3), 267–279 (2007)
20. Artho, C., Havelundand, K., Biere, A.: A high-level data race. Journal on Software Testing, Verification & Reliability (2003)
21. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2008)
22. Shi, Y., Park, S., Yin, Z., Lu, S.: Do I use the wrong definition DeFuse definition-use invariants for detecting concurrency and sequential bugs. In: Proceedings of the OOPSLA, pp. 160–174 (2010)
23. Narayanasamy, S., Wang, Z., Tigani, J.: Automatically classifying benign and harmful data races using replay analysis. In: Proceedings of the PLDI 2007 Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 22–31 (2007)
24. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2006) (2006)

Author Index