LOW OVERHEAD DATA RACE DETECTION TECHNIQUES FOR LARGE OPENMP APPLICATIONS

by

Simone Atzeni

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2017

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Simone Atzeni**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan,	Chair(s)	**08/10/2017** Date Approved
Zvonimir Rakamarić,	Co-Advisor	**08/15/2017** Date Approved
Dong H. Ahn,	Member	**08/07/2017** Date Approved
Hari Sundar,	Member	**08/10/2017** Date Approved
Ryan Stutsman,	Member	**08/10/2017** Date Approved

by **Ross Whitaker**, Chair/Dean of

the Department/College/School of **Computer Science**

and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

High Performance Computing (HPC) on-node parallelism is of extreme importance to guarantee and maintain scalability across large clusters of hundreds of thousands of multicore nodes. HPC programming is dominated by the hybrid model "MPI + X", with MPI to exploit the parallelism across the nodes, and "X" as some shared memory parallel programming model to accomplish multicore parallelism across CPUs or GPUs. OpenMP has become the "X" standard de-facto in HPC to exploit the multicore architectures of modern CPUs. Data races are one of the most common and insidious of concurrent errors in shared memory programming models and OpenMP programs are not immune to them. The OpenMP-provided ease of use to parallelizing programs can often make it error-prone to data races which become hard to find in large applications with thousands lines of code. Unfortunately, prior tools are unable to impact practice owing to their poor coverage or poor scalability.

In this work, we develop several new approaches for low overhead data race detection. Our approaches aim to guarantee high precision and accuracy of race checking while maintaining a low runtime and memory overhead. We present two race checkers for C/C++ OpenMP programs that target two different classes of programs. The first, ARCHER, is fast but requires large amount of memory, so it ideally targets applications that require only a small portion of the available on-node memory. On the other hand, SWORD strikes a balance between fast zero memory overhead data collection followed by offline analysis that can take a long time, but it often report most races quickly. Given that race checking was impossible for large OpenMP applications, our contributions are the best available advances in what is known to be a difficult NP-complete problem.

We performed an extensive evaluation of the tools on existing OpenMP programs and HPC benchmarks. Results show that both tools guarantee to identify all the races of a program in a given run without reporting any false alarms. The tools are user-friendly, hence serve as an important instrument for the daily work of programmers to help them identify data races early during development and production testing. Furthermore, our demonstrated success on real-world applications puts these tools on the top list of debugging tools for scientists at large.

To Nicole, the love of my life.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My PhD journey has come to an end. Although it has been full of challenges, it has also been intellectually fulfilling and a rewarding experience. In this page, I would like to acknowledge those people who have been part of this journey and have helped me in many ways during my 4 years here at Utah.

First and foremost, I would like to thank my advisor Prof. Ganesh Gopalakrishnan who has been essential during my whole PhD experience. I cannot thank him enough for his constant support during my research experience, which includes not only financial support and research advice, but also emotional support every time my PhD life put me in front of new challenges. Second, I would like to thank my coadvisor Zvonimir Rakamarić who has been more a second advisor rather than a coadvisor. Zvonimir's research advice and emotional support have been instrumental in shaping my whole PhD experience. Third, I would like to truly thank Dong H. Ahn from Lawrence Livermore National Laboratory (LLNL) for his financial support, research advice, and for giving me the opportunity to work in one of the largest computing facilities in the United States. Dong has been for me a third advisor during my PhD experience, he has given me the opportunity to spend two summers at LLNL where I could apply my research in real-world scenarios and collaborate with some of the greatest scientists in the world. Finally, I would like to sincerely thank the rest of my committee members Hari Sundar and Ryan Stutsman, who provided me great guidance and always find the time to help me whenever I was in need.

Thanks to all my friends in the Gauss and Soarlab research groups for their great support. In particular I would like to thank Marko Dimjašević and Mohammed Al-Mahfoudh, who have shared this experience with me since the beginning and together we have faced many of the PhD challenges, classes, but also fun and emotional moments. Thanks to Sriram Aananthakrishnan, Mark Baranowski, Geof Sawaya, Montgomery Carter, Charlie Jacobsen, Ian Briggs, and Shaobo He for all the technical discussions, coding advice, political talks, and pub nights out. All of you made these 4 years easier, fun, and memorable.

I would also like to thank two friends in particular that I have met here at Utah. Thanks to my friend Murry Mullenax, from the Steiner's Master Swimming group, for his advice on technical writing, for helping me out by proofreading this dissertation, and for our endless conversations. I would also like to thank Maurizio Bocca, an Italian fellow that I met at the beginning of my PhD who has become a great friend and gave me great advice for successfully complete my PhD

This dissertation would not have been possible without the support of my family. Thank you, mom Lucia, dad Tonio, and Alessia for the video calls during my lunch breaks, your texts, and your encouragement and emotional support during these 4 years. I missed you all, and I look forward to being able to come visit all of you more often now that my PhD has come to an end.

During my PhD I met the love of my life, Nicole. She and her family have welcomed me in their home, given me emotional support and encouragement every time I needed it the most. Thanks to Nicole's parents, Cherie and Lee, for helping me by proofreading this dissertation, and becoming my second family while being far away from home. Thanks to Nicole's siblings Chris, Bobbi, Carrie, and Gabe for the fun family dinners and for being always there to enjoy with me the holidays and breaks. I feel very lucky to have Nicole in my life, and now that this PhD journey has come to an end we look forward to starting our life together.

CHAPTER 1

INTRODUCTION

Multithreaded programming has become widespread in use, given the need to employ multicore CPUs to gain higher performance at a given energy budget. In the High Performance Computing (HPC) world, this has led to an increased adoption of on–node parallelism in large software applications. The work in progress at national research facilities [1, 2, 3] confirms this trend.

Multithreaded programming is achieved through different programming models (e.g. Pthreads); however, the predominant paradigm of choice in HPC is OpenMP [4], which guarantees portability and ease of use. At the Lawrence Livermore National Laboratory (LLNL), one of the world's largest computing facilities, one of the main ongoing tasks for computational scientists is the porting of critical multiphysics applications [5] to OpenMP. In this community, OpenMP is of paramount importance to enable shared memory parallel programming; yet, porting large HPC applications to OpenMP is error-prone. The correctness of these applications is crucial to the reliability of critical simulations pertaining to real world phenomena of fundamental importance such as modeling of nuclear explosions, weather simulations, hydrodynamics modeling, and so forth. One of the most common error types in OpenMP applications is the data race [6]. Data race detection is hard to achieve with traditional debugging methods, and it is also known to be a NP-complete problem [7]. Fast and precise checking tools to detect data races are needed now more than ever. While data race is a well-known problem and Pthreads data race detection tools have been proposed over the past 20 years, none or just a few of them are actually able to analyze OpenMP programs. This dissertation targets this critical need.

Data race detection research has focused both on static and dynamic analysis techniques. Static analysis techniques allow for exploration of all the inputs of the program and the interleavings of the threads. In addition, they are scalable and fast since they do not incur any runtime overhead. However, the lack of information that exists only at runtime makes these techniques imprecise; in fact, they often miss races and generate false positives. Runtime techniques are precise, as they do not report any false positives, and only report races in the branches of programs that are actually executed. On the other hand, dynamic analysis for data race detection is known to generate a high runtime and memory overhead due to the operations it needs to perform and the state it needs to maintain during the execution.

The runtime overhead of even the best of dynamic tools, such as the ThreadSanitizer (Tsan) and Intel®Inspector XE, can cause between 5×–20× slowdown, and the memory overhead can be between 2×–10× of the memory used by the normal execution of the programs. For large programs, such as HPC applications, the runtime and memory overheads can be even larger. The high runtime slowdown and memory usage make such tools useless from the point of view of actual developers, who probably would not be keen on waiting a long time to check their programs *or they may not even have enough machine resources to run the tools*. We definitely need better techniques – static, dynamic, or combinations – to detect data races in large OpenMP applications.

1.1 Thesis Statement

The goal of this dissertation is to provide new approaches for data race detection with low runtime and memory overheads while maintaining high precision and accuracy. With that said, our thesis statement is the following:

> Combining the best of existing static and dynamic analysis techniques, and tailoring the implementation to the actual concurrency structure of structured parallel languages such as OpenMP, we can make data race checking of HPC applications practical.

1.2 Background

The problem of data race detection has been tackled by several researchers [8, 9, 10, 11, 12, 13, 14], through different techniques. Existing data race detection methods can be classified in four different categories as shown in Figure 1.1. Methods based on static analysis, dynamic analysis, hybrid techniques, and symbolic execution present strengths and limitations that raise the need for developing new techniques to enable or improve

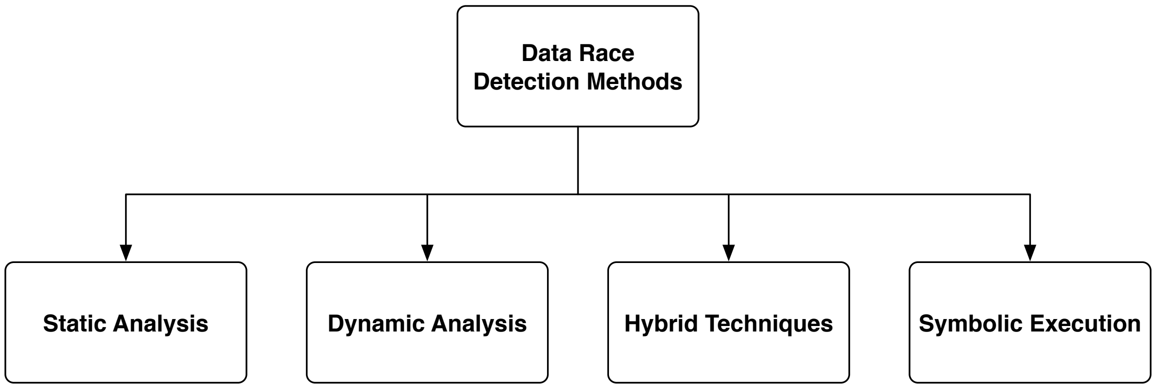


Figure 1.1: Taxonomy of existing data race detection techniques.

the analysis of large OpenMP HPC applications. We explain the pros and cons of existing techniques that belong to the categories we identified:

- *Static Analysis:* Static analysis techniques utilize methods such as dependency analysis [15], alias analysis [16], type systems [17], interprocedural analysis [18], and so forth. While static analyses are able to consider all possible program behaviors without actually executing the program, they may report false positives due to dynamic behaviors that can not be actually modeled (e.g., pointers and aliases). Thus, static analysis is often not effective and can report false alarms or miss races.

- *Dynamic Analysis:* Dynamic techniques rely on compiler or binary instrumentation to gather information at runtime about memory accesses, synchronization operations, thread identifiers, and so forth. Dynamic approaches base their analysis on techniques such as happens-before [11, 9] or lockset analysis [10]. These techniques are more accurate; however, they depend on the current execution of the program, and in the case where a date race does not exhibit itself for a specific run, they miss the race. In addition, the overhead is often prohibitively high, which makes them inappropriate to analyze large scale applications.

- *Hybrid Techniques:* Hybrid techniques combine static and dynamic approaches. Static analyses are often used to collect information that can be used by the dynamic analyses to increase the precision or reduce the overhead [14].

- *Symbolic Execution:* Symbolic execution [19] methods try to explore all possible program paths through symbolic values. The execution is then encoded into first-order logic formulas and followed by Satisfiability Module Theories (SMT) based solving. To obtain scalable analysis [20, 21] the encoding must be well designed and optimized based on domain knowledge of the program.

1.3 Contributions of the Dissertation

In this dissertation we present different contributions to overcome the limitations of the aforementioned methods. First, we combine existing techniques such as static and dynamic analyses, to bring together the best of the two approaches which are respectively low overhead and high precision and accuracy. Secondly, we formally define the concurrency of the OpenMP programming model through an operational semantics that exploits the OpenMP concurrency structure for race detection. Finally, starting from the operational semantics definition, we implement a novel OpenMP data race detection technique which guarantees zero memory overhead, soundness, and completeness of the data race detection analysis for a given input. These contributions result in two different data race detection tools for OpenMP programs that we present in detail in the chapters that follow.

The first tool, ARCHER (see Chapter 2), applies static analysis techniques to identify race free regions of code and remove them from the runtime analysis. On the other hand, the dynamic analysis checks the rest of the program for data races by applying an happens-before based technique. The results of our work show high precision and accuracy while maintaining a low runtime overhead. However, ARCHER suffers from high memory overhead ($6\times$) memory overhead, which makes it unsuitable for large applications that require more than 16% of the available memory.

The high memory overhead issue inspired us to research and implement a new technique to reduce the memory overhead. Therefore we created SWORD, another OpenMP data race checker (Chapter 4) based on a formal operational semantics definition that we explain in Chapter 3. SWORD implements a fast logging technique to save information about the program memory accesses into files. This approach keeps the memory overhead to almost zero. We then implemented an offline analysis technique of the logs to identify the races. The offline analysis is highly parallelizable both across a multicore architecture and across a cluster. Results show that the logging techniques plus offline race detection algorithm reduce the memory overhead, enabling the analysis of large HPC applications that were not possible with the existing tools.

The new approaches are novel contributions to effective and efficient data race detection for OpenMP programs. The two tools, subjects of this dissertation, are complementary in order to cover different classes of OpenMP programs. ARCHER is fast and detects most of the data races; however, because of its high memory overhead, it can only analyze a class of applications that requires a small amount of memory to complete the data race detection process. On the other hand, SWORD provides a technique that is able to analyze programs that necessitate of large amount of memory where other tools would fail, and thus *guaranteeing better coverage for a given run*.

1.4 Organization of the Dissertation

This dissertation is organized as follows: In Chapter 2, we present the first tool ARCHER with details about its data race detection approach and an extensive evaluation on well-know benchmarks and real-world applications; Chapter 3, illustrates an operational semantics to enable precise and accurate data race analysis exploiting the structured concurrency model of OpenMP; with Chapter 4, we present our implementation of the operational semantics in a tool called SWORD, with experimental results that demonstrate the effectiveness and efficiency of the approach; finally in the Chapter 5, we summarize all the contributions and conclude the dissertation.

CHAPTER 2

EFFECTIVELY SPOTTING DATA RACES IN LARGE OPENMP APPLICATIONS

This chapter is based on work published at the Workshop on the LLVM Compiler Infrastructure in HPC [22] and at the IEEE International Parallel & Distributed Processing Symposium 2016 [23][1].

In this chapter we present ARCHER, the first of the two data race detection tools subject of this dissertation. We illustrate the details of ARCHER's techniques and an extensive evaluation of the tool in term of effectiveness and efficiency.

2.1 Introduction

High performance computing (HPC) is undergoing an explosion in raw computing capabilities as evidenced by recent announcements of next-generation computing system projects [1, 2, 3]. To meet the stipulated performance and power budgets, many key software components in these projects are being transitioned to adopt on-node parallelism to a greater degree. The predominant programming model of choice in this transition is OpenMP—due in large part to its portability and ease of use. In fact, the main task for computational scientists at Lawrence Livermore National Laboratory (LLNL), one of the world's largest computing facilities, is the porting of mission-critical multiphysics applications [5] to exploit OpenMP.

We find, however, that efficient and scalable development tools for OpenMP are still quite scarce, making development efforts hard. In particular, none of the preexisting OpenMP data race checkers is capable of handling the code sizes involved, or provides effective debugging support for concurrency bugs. Meanwhile, libraries such as Hypre [24],

[1]©2016 IEEE. Adapted, with permission, from Atzeni S., Gopalakrishnan G., Rakamarić Z., Ahn D. H., Laguna I. Schulz, M, Lee G., Protze J., Müller M. S., Archer: Effectively Spotting Data Races in Large OpenMP Applications, June 2016

which underlie many critical applications, have run into data races during this transition. In one LLNL application, because of this lack of debugging tools, developers who faced these races even took the draconian approach of reverting back to sequential code.

This work describes ARCHER, our new OpenMP data race detector, its unique capabilities in terms of scalability and precision, its use of a proposed standard, and our philosophy of building on well-engineered open-source software. While the core concept of a data race has been known for decades (uncoordinated, i.e., not separated by a happens-before edge, accesses on a memory location by two threads, with one access being a write), transitioning this idea into HPC practice required adherence to four key tenets.

1. *Scalable Happens-Before Tracking Methods:* Checking for races in production OpenMP programs requires the ability to track a huge number of memory references and their happens-before ordering. A significant amount of ARCHER's scalability stems from its exploitation of a preexisting tool—namely ThreadSanitizer (TSan) [12]. TSan's unique architecture enables it to implement the idea of vector-clock-based race checking far more efficiently than comparable tools do. Embracing TSan and its LLVM-based tooling approach enables us to write custom LLVM passes, and in general take advantage of the growing popularity of LLVM in HPC [22]. Previous OpenMP data race checking tools were never released for public evaluation, were based on binary instrumentation through PIN [25], or employed symbolic methods [20]. These approaches are neither scalable nor widely portable. ARCHER has been publicly released under the BSD License [26]. (Note: TSan was originally designed for PThread and Go programs, and cannot be directly applied to OpenMP programs as will soon be described.)

2. *Static/Dynamic Analysis of Structured Parallelism:* In ARCHER, we capitalize on OpenMP's structured parallelism to support two key features never before exploited in an OpenMP race checker. First, we exploit OpenMP's structured parallelism to easily write LLVM passes that identify guaranteed sequential regions within OpenMP. Such analysis would be difficult to conduct in the context of unstructured parallelism (e.g., PThreads). Second, we identify and suppress parallel loops from race checking. ARCHER achieves this by black-listing accesses within parallelizable loops with the help of a static

analysis.

3. *Modular Interfacing with OpenMP Runtimes:* While structured parallelism has been exploited in the context of Java-like languages (e.g., Habanero Java [27]), such exploitation in the context of OpenMP and ARCHER required a combination of innovations. Unlike in languages such as Habanero Java where the language and the runtime are designed together, in OpenMP vendors provide their own custom runtimes. Tools, such as TSan, must be suitably modified to ignore OpenMP internal actions, which may otherwise be falsely assumed to be data races [22]. ARCHER's approach is architected based on the OMPT standard [28] so that our solutions may modularly be incorporated with multiple OpenMP runtimes.

4. *Collaboration with Active Projects:* ARCHER has already made significant impact within LLNL. As one example, HYDRA [29] is a large multiphysics application developed at LLNL, which is used for simulations at the National Ignition Facility (NIF) [30] and other high energy density physics facilities. It comprises many physics packages (e.g., radiation transfer, atomic physics, and hydrodynamics), and although all of them use MPI, a subset of them use thread-level parallelism (OpenMP and PThreads) in addition to MPI. It has over one million lines of code and a development lifetime that exceeds 20 years. In the summer of 2013, developers began porting HYDRA to Sequoia [31], the over 1.5 million core IBM Blue Gene/Q-based system that had just been brought online at that time. Although the efforts included incorporating more threading for performance, the developers got significantly impeded when they could not resolve a non-deterministic crash on an OpenMP-threaded version of Hypre [24] (used by one of HYDRA's scientific packages). The developers found it very difficult to debug this error that occurred intermittently after varying numbers of time steps, only at large scales (at or above 8192 MPI processes), and only under compiler optimizations. After spending considerable amounts of time, the team suspected the presence of a data race within Hypre, but the difficulties in debugging and time pressure forced them to work around the issue by selectively disabling OpenMP in Hypre. When ARCHER was brought onto the scene, it located "benign races" involving two threads racing to write the same value to the same location—a practice

known to be dangerous in the presence of compiler optimizations [32]. Removing these benign races fixed the bug. This episode—detailed in Section 2.3.3—clearly shows that effective data race checkers specifically tailored to high-end computing environments are invaluable during critical projects.

2.2 Approach

Figure 2.1 illustrates how ARCHER implements our tenets by combining well-layered modular static and dynamic analysis stages. In more detail, our static analysis passes [33, 34, 35] help classify the given OpenMP code regions into two categories: *guaranteed race-free* and *potentially racy*. Our dynamic analysis then applies state-of-the-art data race detection algorithms [12, 9] to check only the potentially racy OpenMP regions of code. The static/dynamic analysis combination is central to the scalability (while maintaining analysis precision) of ARCHER, as evidenced by its ability to handle real-world examples that existing tools cannot handle with the same levels of precision and scalability (see Section 2.3.2).

As described earlier, we implemented ARCHER using the LLVM/Clang tool infrastructure [36, 37] and the TSan dynamic race checker [12]. On the static analysis side, ARCHER uses Polly [35] to perform data dependency and loop-carried data dependency analysis (together called *data dependency analysis* from now on). This results in a *Parallel Blacklist*. ARCHER also extends some of the static analysis passes already present in LLVM. Specifically, our extension builds a call graph and traverses it to identify memory accesses that do not come from within an OpenMP construct (i.e., sequential code regions). This results in a *Sequential Blacklist*. These blacklists are combined and used to limit the instrumentation in TSan.

On the dynamic analysis side, ARCHER uses our customized version of TSan to detect

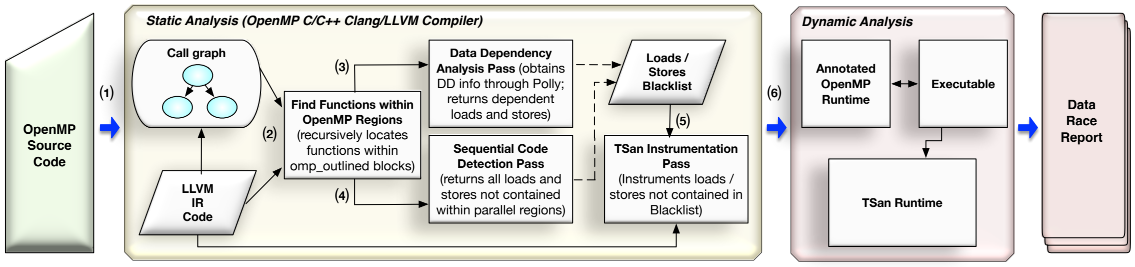

Static Analysis (OpenMP C/C++ Clang/LLVM Compiler)
OpenMP Source Code
(1)
Call graph
LLVM IR Code
(2)
Find Functions within OpenMP Regions (recursively locates functions within omp_outlined blocks)
(3)
(4)
Data Dependency Analysis Pass (obtains DD info through Polly; returns dependent loads and stores)
Sequential Code Detection Pass (returns all loads and stores not contained within parallel regions)
Loads / Stores Blacklist
(5)
TSan Instrumentation Pass (Instruments loads / stores not contained in Blacklist)
(6)
Dynamic Analysis
Annotated OpenMP Runtime
Executable
TSan Runtime
Data Race Report

Figure 2.1: ARCHER tool flow.

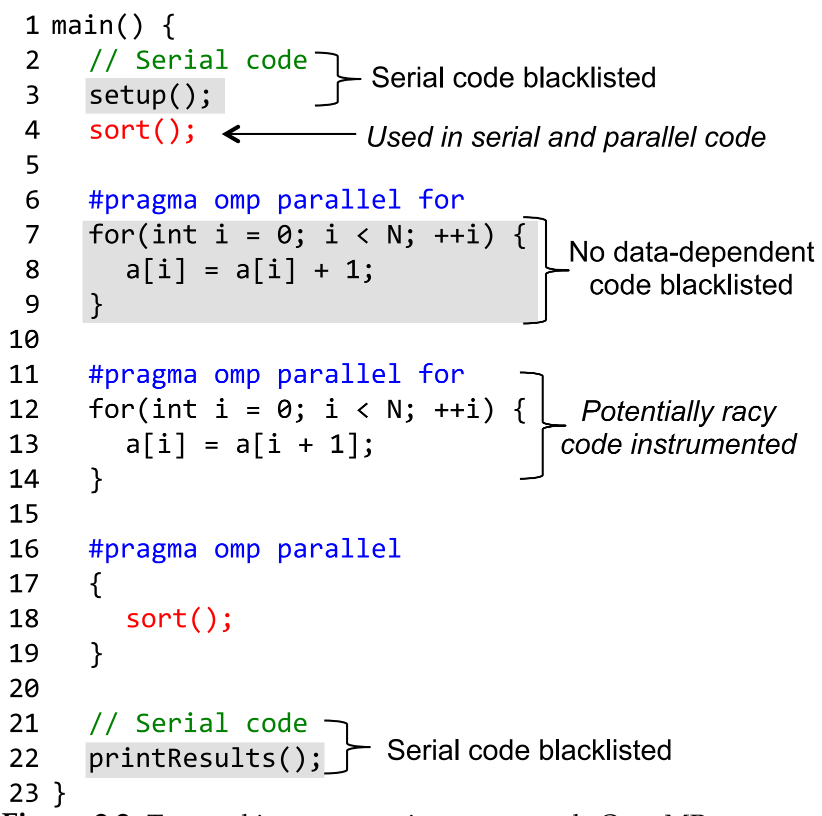
data races at runtime. To prevent TSan from being confused by OpenMP runtime internal actions (and falsely report them as OpenMP-level races), ARCHER employs TSan's Annotation API to highlight these synchronization points within LLVM OpenMP Runtime (the runtime presently associated with ARCHER in our studies). As we have already pointed out, our efforts are being migrated to adhere to the OMPT standard.[2]

2.2.1 Static Analysis Phase

We now detail some of the finer details of our static analysis, including feeding the blacklist information to the TSan runtime. TSan carries out its dynamic data race detection by first instrumenting all the load and store actions of a program at compile time, and using this instrumentation to help track happens-before. TSan also provides a feature that allows users to blacklist functions [38] (by their name) that should not be instrumented and that are thus to be ignored at runtime. Unfortunately, this granularity of instrumentation is insufficiently refined to handle our sequential and parallel blacklists that express the intent to blacklist individual accesses (that are, in almost all cases, not demarcated by function boundaries). Thus, in order to communicate our blacklists to TSan, we extended its blacklisting capabilities to enable a finer-grained selection at the level of source lines. This allows the modified TSan used by ARCHER to exploit our sequential and parallel blacklists, thus guaranteeing a high degree of analysis precision and scalability.

In more detail, after the LLVM intermediate representation (IR) and call graph are generated, our analysis transforms OpenMP pragmas in the LLVM IR code as outlined functions named *omp_outlined.NUM*, where *NUM* is an identifier for each parallel region present in the code. Our first pass visits the call graph, and for each *omp_outlined* function finds all the functions called within it. For each of these functions, the analysis is recursively applied. Thereafter, data dependency analysis and sequential code detection are applied (step (3) in Figure 2.1). For the former, an existing tool in the LLVM/Clang suite called Polly [35] is used. In the example given in Figure 2.2, the first for-loop (lines 7–9) is data parallel (i.e., data independent) and is blacklisted, while the second one (lines 12–14) is not (exhibits a loop-carried dependence) and hence is not blacklisted.

[2]Some of us are associated with the OMPT efforts, thus facilitating our collaboration further to benefit a wide variety of OpenMP runtimes.



```
1  main() {
2    // Serial code
3    setup();
4    sort();
5
6    #pragma omp parallel for
7    for(int i = 0; i < N; ++i) {
8      a[i] = a[i] + 1;
9    }
10
11   #pragma omp parallel for
12   for(int i = 0; i < N; ++i) {
13     a[i] = a[i + 1];
14   }
15
16   #pragma omp parallel
17   {
18     sort();
19   }
20
21   // Serial code
22   printResults();
23 }
```

Serial code blacklisted
Used in serial and parallel code
No data-dependent code blacklisted
Potentially racy code instrumented
Serial code blacklisted

Figure 2.2: Targeted instrumentation on a sample OpenMP program.

ARCHER also identifies sequential code sections (step (4)). In Figure 2.2, lines 3 and 22 are sequential instructions and are hence blacklisted. However, function `sort()`, invoked at lines 4 and 18, cannot be blacklisted, as it is invoked both from a sequential and parallel context. The payoff due to such sequential code detection is potentially very high in real-world projects where only some of the loops are parallelized with OpenMP (based on the benefits, the number of cores available, etc.). As already pointed out, these analyses are greatly facilitated by OpenMP's structured parallelism.

2.2.2 Dynamic Analysis Phase

Our use of TSan for OpenMP race checking hinges on the fact that OpenMP parallelism is typically realized through a PThread-based runtime library. As already mentioned, unmodified TSan cannot be meaningfully used for OpenMP due to the large number of false positives ("false alarms") it reports [22].

The OpenMP standard specifies several high-level synchronization points. Explicit synchronization points include `barrier`, `critical`, `atomic`, and `taskwait`. Implicit synchronization includes `single`, `task`, and the OpenMP `reduction` clause. As semantically intended and realized in the runtime, the threads can enter a critical section only in a serialized manner, thus avoiding a data race. However, TSan lacks any knowledge about these synchronization points. We use the Annotation API of TSan to mark these synchronization points within the OpenMP runtime to avoid such false positives. This technique was successful in eliminating all false positives in our benchmarks. Finally, the combination of the ARCHER's static analysis and new TSan instrumentation that exploits the blacklist information produces a selectively instrumented binary (step (6) in Figure 2.1).

2.3 Evaluation

We evaluate ARCHER in three stages through: (1) a collection of smaller benchmarks called the OmpSCR benchmark suite [39] (an OpenMP source code collection); (2) AMG2013, a non-trivial application from the HPC CORAL benchmark suite [40]; and (3) the HYDRA case study. Our evaluation is in terms of the effectiveness, performance, and scalability of ARCHER compared to Intel®Inspector XE. We also compare ARCHER against an unmodified version of TSan applied to the same benchmarks.[3] When using TSan and ARCHER, we compile our benchmarks using Clang/LLVM, and when using Intel®Inspector XE, we compile them using the Intel Compiler. When running our benchmarks under ARCHER, we link them against our annotated LLVM OpenMP Runtime [41, 22]. When running them under Intel®Inspector XE as well as TSan, we employ the uninstrumented version of the same runtime. We studied the following configuration selections:

1. **ARCHER:** We employ four configurations: (1) the basic configuration of ARCHER that applies both static and dynamic analysis to reduce runtime and memory overhead; (2) ARCHER run without static analysis support (only dynamic race checking using the enhanced runtime to avoid false positives is used); (3) apply just the Sequential Blacklist; and (4) apply just the Parallel Blacklist.

2. **TSan:** When running the unmodified version of TSan, we employ its default parameters.

[3]Despite this exercise yielding numerous false positives, it provides a good performance baseline.

3. **Intel®Inspector XE:** Intel®Inspector XE provides many "knobs" for controlling performance and analysis quality tradeoffs. Of these, we exercise three configurations: (1) a *default* mode that checks memory accesses at the coarse-grain granularity of four bytes; (2) the *extreme-scope* configuration that sets memory access granularity at a single byte (incidentally, this is the same granularity as what TSan employs), which obtains higher precision at higher cost; (3) the *use-maximum-resources* configuration that allows Intel®Inspector XE to detect more data races, but at the cost of increased memory consumption and greater runtime overhead.

We perform our evaluation on the Cab cluster at LLNL. Each Cab node has two 8-core, 2.6 GHz Intel Xeon E5-2670 processors and 32GB of RAM. It runs the TOSS Linux distribution (kernel version 2.6), which is a customized distribution specifically targeting engineering and scientific applications. Runtimes and memory overhead of all benchmarks were averaged across 10 executions, each time running with a variable number of threads (ranging from 2 to 16). In the experimental results, *Release* denotes the original benchmark characteristics. *SequentialBlacklisting* and *ParallelBlacklisting* denote that just those blacklisting strategies are exploited, ARCHER denotes that both are used, while ARCHER *"no SA"* denotes that none are used.

2.3.1 OmpSCR Benchmark Suite

We chose the OmpSCR benchmark suite (see Table 2.1) primarily because it harbors a few known races, as reported in prior work [25]. We, however, found several additional races not previously reported. With respect to each tool and configuration, we now describe the overall analysis quality followed by the runtime overheads. Then, we summarize the overall merit of these tools by plotting their analysis quality vs. performance scores.

Our evaluation shows that ARCHER detects all of the documented races in all configurations. In particular, it discovered six such races in the following benchmarks: `c_md`, `c_loopA.badSolution`, `c_loopB.badSolution1`, `c_loopB.badSolution2`, `c_testPath`, and `c_jacobi3`. In addition, ARCHER reported six previously undocumented races in the following C++ benchmarks: `cpp_qsomp1`, `cpp_qsomp2`, `cpp_qsomp3`, `cpp_qsomp4`, `cpp_qsomp5`, and `cpp_qsomp6`. (We manually verify that all the reported races are real.) In contrast,

Table 2.1: Execution slowdown factor for various tool configurations.

Benchmark	InspectorDefault	InspectorExtremeScope	InspectorMaxResources	ARCHER "no SA"	ARCHER
c_fft	18.2	22.1	66.8	8.1	7.9
c_fft6	21.0	25.3	188.8	12.2	12.7
c_jacobi01	38.2	27.5	25.2	19.2	15.6
c_jacobi02	38.7	26.7	25.6	19.6	17.8
c_loopA.badSolution	5.1	7.0	41.1	5.9	3.9
c_loopA.solution1	10.2	12.2	64.9	9.4	10.5
c_loopA.solution2	5.1	7.1	41.2	5.5	3.6
c_loopA.solution3	4.5	5.8	42.1	5.1	4.2
c_loopB.badSolution1	6.2	7.5	36.8	5.5	3.8
c_loopB.badSolution2	15.6	16.2	43.7	2.3	2.3
c_loopB.pipelineSolution	5.4	7.8	36.7	5.6	3.6
c_lu	18.0	19.6	240.7	13.8	13.0
c_mandel	5.6	5.4	5.3	1.7	1.7
c_md	12.7	21.1	253.4	197.3	197.1
c_pi	11.1	10.7	11.1	2.3	2.6
c_qsort	14.2	16.9	34.1	5.8	5.7
c_testPath	133.0	133.6	138.3	18.3	17.9
cpp_qsomp1	57.5	57.4	289.5	18.0	18.1
cpp_qsomp2	57.8	57.6	286.6	17.9	11.9
cpp_qsomp5	56.8	62.5	338.2	20.4	20.8
cpp_qsomp6	57.5	57.9	253.5	18.2	11.9
cpp_qsomp7	57.8	57.8	229.3	18.8	18.3
Mean	29.5	30.3	122.4	19.6	18.4
Median	16.8	20.3	54.3	10.8	11.2
Geometric Mean	18.3	20.2	71.5	10.0	8.8

Intel®Inspector XE incurs varying degrees of accuracy and precision loss in all three configurations.

In term of accuracy (the number of correctly detected races divided by the number of true races that should have been detected), Intel®Inspector XE, in its default and extreme-scope configurations, misses races in benchmarks `c_loopB.badSolution1`, `cpp_qsomp1`, `cpp_qsomp2`, `cpp_qsomp5`, and `cpp_qsomp6`. On the other hand, Intel®Inspector XE under the max-resources configuration detects most of these races, though it still misses the races in `cpp_qsomp5` and `c_loopB.badSolution1`.

In terms of precision (the number of correctly detected races divided by the number of all the races detected, including false positives—i.e., “false alarms”), ARCHER in both configurations[4] incurs no false positives, while Intel®Inspector XE does. For example, in benchmark `c_md`, Intel®Inspector XE reports an additional race that is clearly a false positive, as documented in related work [42]. In addition, in `cpp_qsomp7`—which uses the tasking construct as per OpenMP 3.1—Intel®Inspector XE reports two false positives, which ARCHER in both configurations correctly avoids reporting as races. These results clearly demonstrate that ARCHER accurately understands the OpenMP task synchronization semantics.

We now discuss in detail performance results in terms of runtime and memory overheads. We only present the results for 16 threads because the tools incur similar overheads as we vary the number of threads.[5] Figure 2.3 details runtime and memory overheads for benchmarks in the OmpSCR benchmark suite. In a nutshell, ARCHER outperforms Intel®Inspector XE across all of its configurations on most of the benchmarks. Intel®Inspector XE incurs the least overhead in its default configuration, but this comes at the expense of degraded analysis quality. The *extreme-scope* configuration of Intel®Inspector XE, which is closer to the ARCHER's analysis granularity, incurs much higher overhead than ARCHER with a few exceptions. The *max-resources* configuration results in a very high resource consumption and its overheads are always higher than that of ARCHER.

ARCHER performs slightly better with static analysis support than without, catching all the data races in both cases. This can be mainly attributed to the fact that the OmpSCR benchmarks are small (in terms of the lines of code), and hence static analysis finds very few blacklisting opportunities. Still, ARCHER with static analysis support is overall 15% faster on the average. In Section 2.3.2, we show that on real-world HPC application static analysis reduces much more the runtime overhead, thus underscoring its importance in practice.

[4]We omit the evaluation of ARCHER in the Sequential and Parallel Blacklisting configurations for the OmpSCR benchmark suite since the results for those configurations match the results of ARCHER without static analysis.

[5]We omit three OmpSCR benchmarks in our performance results. The data race in `c_jacobi3` highly influences the execution time of the benchmark, varying it by a factor of 1000 from run to run. The other two are `cpp_qsomp3` and `cpp_qsomp4`, where data races cause them to crash.

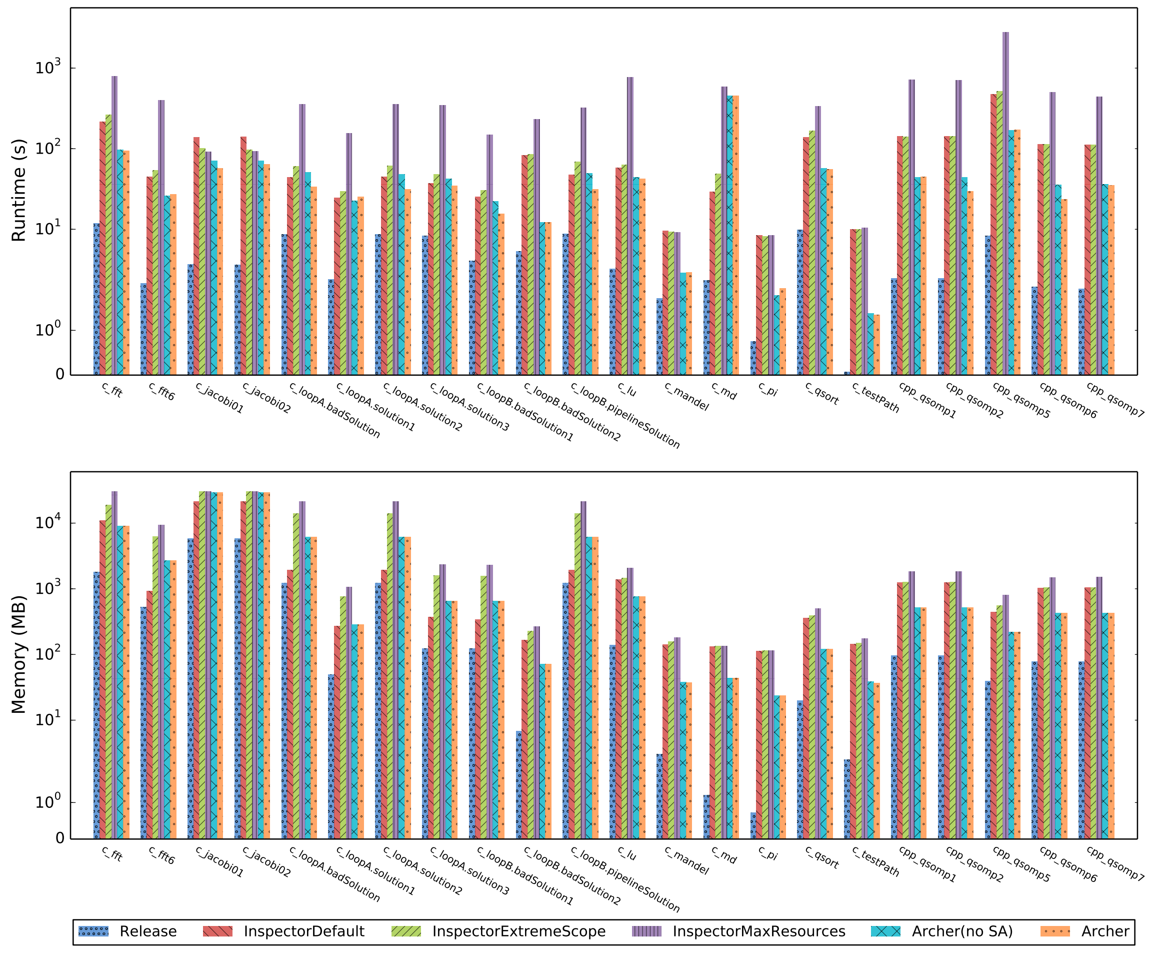
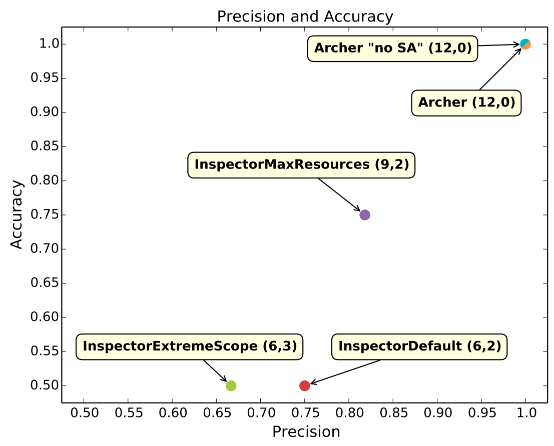
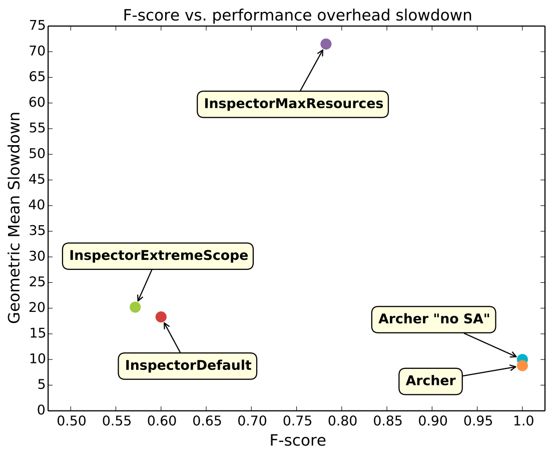


Figure 2.3: Runtime and memory overhead of the tools on the OmpSCR benchmark suite executed with 16 threads.

We assess the merits of the tools by plotting their analysis quality against performance. Table 2.1 shows the execution slowdown for each of the OmpSCR benchmarks under each of the tool configurations. We give the mean, median, and geometric mean in the last three rows. For space reasons, we omitted our other statistical measurements. However, using a confidence level of 0.05, we compared the slowdown distributions of each configuration (i.e., how our 10 measurements varied for each target benchmark) and verified that the distributions of ARCHER and ARCHER "no SA" do not overlap for a majority of cases. This indicates that the difference in performance between ARCHER and ARCHER "no SA" is statistically significant. In addition, Figure 2.4a gives the precision and accuracy of the tools, displayed with their true and false positives counts. The plot show that ARCHER provides the best analysis quality with respect to other state-of-the-art race detectors in-



(a) Precision and accuracy; in parentheses we report the number of reported races and false positives.



(b) Overall merit expressed as analysis quality vs. performance.

Figure 2.4: Different metrics of comparison for the tools.

cluding Intel®Inspector XE.

In Figure 2.4b, we use an F-score (F1 score) [43] to capture the overall quality of analysis. The F-score is a measure of analysis quality that accounts for both accuracy and precision (as defined previously) and is given by:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{accuracy}}{\text{precision} + \text{accuracy}}.$$

Thus, the F-score reaches its best value at 1 and worst at 0. In Figure 2.4b we plot each tool onto a two-dimensional space defined by the F-score and slowdown geometric mean. We use the geometric mean as our performance metric because the mean and median are significantly skewed by outliers. Indeed, the slowdown values run the gamut from 253.4x to 1.7x (mainly because of the very different charateristics and running times of the OmpSCR benchmarks), and this biases the arithmetic mean and median, while the geometric mean is designed to compute a figure of merit under such circumstances. The plot shows the general attributes of each tool in terms of accuracy and runtime overheads, and our design goal is to create a tool that lies as close as possible to the lower right corner. It is clear from the plot that ARCHER best meets this goal, as compared to other state-of-the-art tools: both versions of ARCHER (with and without static analysis) do much better than Intel®Inspector XE in all its configurations.

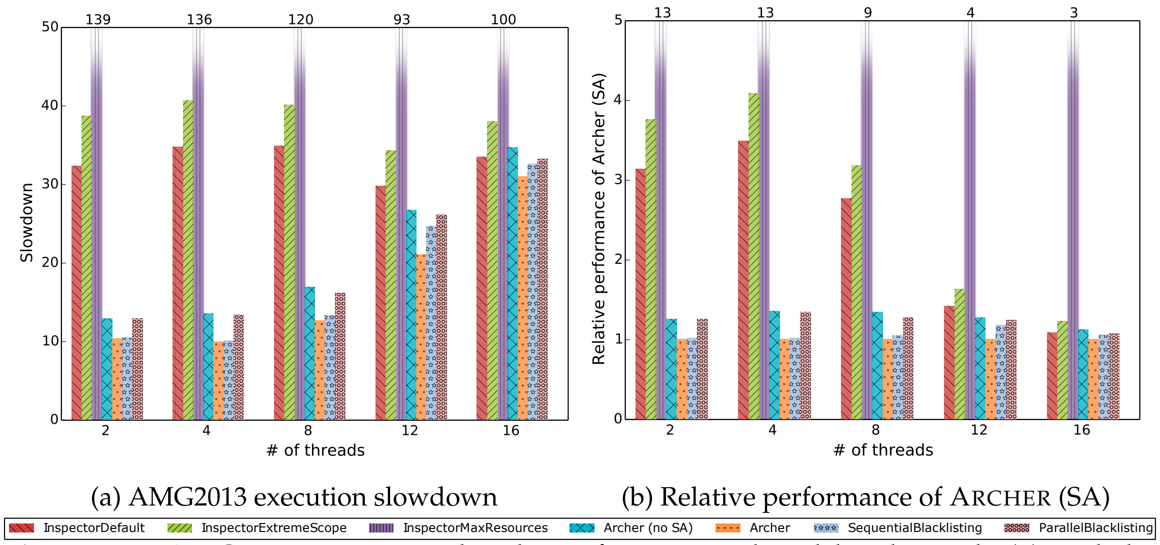
2.3.2 AMG2013 Case Study

To complement our OmpSCR study with a larger code base, we perform an evaluation on AMG2013, which contains approximately 75,000 lines of code. AMG2013 [44] is a parallel algebraic multigrid solver for linear systems and is based on Hypre [24, 45], a large linear solver library developed at LLNL. Our experiments with ARCHER discovered three races within AMG2013, which were previously unreported. Thus, this application was useful to quantify both the performance and analysis quality of the tools. In the following, we compare the precision and performance of unmodified TSan, each ARCHER configuration, and Intel®Inspector XE in three different configurations.

The unmodified TSan, after reporting about 150 false positives, crashes and never finishes its analysis. Intel®Inspector XE reports all three data races when it is configured with *use-maximum resources*. When using the *extreme-scope* configuration, it reports all three of the races, but only when running with 16 threads. Finally, when using the *default* configuration, Intel®Inspector XE always misses one particular race of the three.

We now compare the performance of these tools in all of the different configurations. Figure 2.5 shows the AMG2013 execution slowdown factor introduced by the tools a and the relative performance factor of ARCHER (SA) against the other tools b. Both ARCHER and Intel®Inspector XE are dynamic checkers, and hence they introduce a large runtime overhead with respect to the application execution under no tool control (see Figure 2.5a). However, it is clear that ARCHER has significant performance advantages relative to other tools. In fact, Figure 2.5b shows the relative performance of ARCHER (with and without static analysis) against all of the three configurations of Intel®Inspector XE. ARCHER is generally 2–15x faster than Intel®Inspector XE depending on the number of threads. When compared to itself, ARCHER without static analysis support improves the performance by a factor between 1.2 and 1.5 depending on the number of threads.

ARCHER also reduces the memory overhead relative to Intel®Inspector XE in comparable configurations (modes other than default). However, its memory footprint still appears unnecessarily large. We surmise that this is because of TSan's runtime shadow memory allocation policy, which ARCHER inherits unmodified. In particular, when an array is initialized, all of its elements are accessed, and this causes TSan to allocate shadow memory corresponding to the entire array during initialization. Thereafter, TSan does



(a) AMG2013 execution slowdown (b) Relative performance of ARCHER (SA)

Figure 2.5: AMG2013 execution slowdown factor introduced by the tools (a) and the relative performance factor of ARCHER (SA) against the other tools (b).

not selectively deallocate this shadow memory, for instance, based on whether the array locations are live beyond a certain point. In our future work, we plan to confirm this, and then achieve selective deallocation—a possibility suggested by OpenMP's structured parallelism model.

The overall gains due to static analysis depend on the proportion of sequential regions (for Sequential Blacklisting) and data independent loops (for Parallel Blacklisting). Our future work will focus on characterizing these gains across many more large case studies.

2.3.3 ARCHER Resolves Real-World Races

We now present how ARCHER aided LLNL scientists in resolving the intermittent crashes in HYDRA mentioned in Section 2.1. This investigation was spurred into action when our AMG2013 experiment discovered the three races mentioned earlier. Of the three data races flagged by ARCHER, two[6] were found in a fairly complex OpenMP region spanning over 400 source lines with tens of reaching variables. The fact that these flagged sites were contained within a deeply nested control-statement level further complicated manual analysis; thus, we contacted the developer for further validation.

In response, the developer confirmed that both were indeed true races. Specifically,

[6]Specifically, one between the memory accesses at lines 1183 and 1248 and the other at lines 1184 and 1249 within `par_interp.c`

one thread accesses the first element of a portion of an array defined by `P_diag_i` and `P_offd_i` (belonging to the next thread), while the second thread subtracts a number from this element. However, because the number being subtracted for this particular element was zero, this condition was never detected during testing. While programmers often consider this type of races (i.e., multiple threads writing the same value to the same memory location) benign, the developer did recognize that the containing function, `hypre_BoomerAMGInterpTruncation`, was one of the routines that they had to disable OpenMP parallelization on for reliable use within HYDRA.

Encouraged by our findings, the application's team resumed their debugging of this issue. They applied a fix to these benign data races to the latest Hypre release (2.10.0b) and reran the simulation. This time, however, the simulation failed in a different way: a crash occurred very quickly and much more deterministically. Next, we applied ARCHER to this Hypre release using a representative test code provided by the developer, and ARCHER reported several additional benign data races; tqhe races were detected between lines 2313 and 2315 of `par_coarsen.c` where threads write the constant 0 to the same element in an array: `CF_marker[j] = 0`.

The developer was initially skeptical that these races were the root cause because threads write the same numerical constant: 0 in the coarsening and 1 in `par_lr_interp.c`. However, when we fixed all of these races, for example by synchronizing the respective assignments with OpenMP critical, the crashes no longer appeared. We theorize that the compiler (IBM XL) used on this platform, which would assume race-free code for optimization, transformed the code in such a way that those benign races turned into harmful ones, a pitfall described previously by Boehm [32].

While the developer is currently trying to find a way to resolve the races in a more performant manner, it was made clear that data race checkers like ARCHER, which are tailored to large HPC applications, are crucial to avoid a programmer productivity loss on such elusive bugs.

2.4 Related Work

Data race detection in general is one of the most widely studied problems in concurrent program design and has been shown to be NP-hard [7]; a complete survey is beyond the scope of this section and so we focus on closely related approaches for correctness checking.

According to Erickson et al. [46], data races must be taken as "the smoking gun" for any number of root causes: insufficient atomicity (as per intended code behavior), an unreliable communication idiom, unintended sharing [13], or a misunderstanding of how generated code behaves vis-a-vis the higher level program view (including possible mis-compilation [32]). Static race detection methods provide high checking efficiency, but are known to generate false positives (e.g. [47, 8]); each false positive can be a month of wasted reconfirmation time [48]. Polynomial-time race checking can often be achieved under structured concurrency [27]. Predictive methods attempt to find many more "implied" races based on an initial execution through the program (e.g., [49]).

ARCHER derives much of its efficiency by avoiding the instrumentation of independent loop iterations as well as sequential code regions. These approaches to achieve parsimonious instrumentation have recently been shown effective in the context of TSan and PThread programs through a technique called section-based program analysis [50]. The idea of specializing race checking has also taken root in the context of GPU programs where symbolic methods coupled with the idea of using a two-thread abstraction scheme have become popular [21, 51, 52]. This approach is also, in principle, applicable to OpenMP data race checking [20].

2.5 Discussion

Despite OpenMP being around for over two decades, there are no practical data race detectors for OpenMP programs that an HPC practitioner can use in the field today; ARCHER is the first such race checker and its approach is both timely and necessary to provide the widening field of OpenMP programming with this critical correctness tool capability. In fact, the main developer of TSan has taken active interest in our work, and even the LLVM community has helped us by supporting TSan on the PowerPC platform [53].

While ARCHER has proven to be useful at debugging real-world races in OpenMP applications, we now discuss the practical implications of our approach with respect to (1) features in the latest OpenMP specifications and (2) the use of compiler optimization flags.

2.5.1 Latest OpenMP Specifications

The OpenMP Architecture Review Board released the latest OpenMP specification (Version 4.0) in July 2013. We expect that it will take major compilers a few years to come to full compliance with this specification. At the point of writing, there exists no compiler that can fully support OpenMP 4—including its device construct. The OpenMP branch of Clang/LLVM, under which we demonstrate our approach, supports only OpenMP 3.1. While this practical limitation only allowed us to explore the problem space in OpenMP 3.1, we recognize that OpenMP 4, when implemented by compilers and thus adopted by our applications, will present a new set of challenges to our approach.

In particular, with the device construct, OpenMP threads will be run not only on CPUs but also on accelerators, such as GPUs, which could also be subject to the harmful effects of data races. Unfortunately, tooling in this area is not as comprehensive as designers may like. For example, the CUDA Memcheck tool [54] is limited in that it can only detect data races that occur in the thread-block level shared memory space; yet, in practice, races also occur in the global memory scope [52]. Given the current trends to provide coherent memory between CPUs and GPUs [2], it is clear that the community will need more comprehensive race detection techniques. In addition, because of the higher numbers of OpenMP threads that can run on GPUs, techniques to further enhance scalability (e.g., by exploiting thread symmetry relationships) must be researched and developed.

2.5.2 Compiler Optimization Flags

Recent work [32] suggests that it is critical to pinpoint and fix data races that many programmers consider benign. In particular, the presence of any data race can lead a compiler to turn a benign race into a harmful one, even when code transformations that are considered safe are used. In this regard, ARCHER can best detect data races at the source level with no compiler optimization (-O0). This is because an optimization can hide the presence of a race through transformations. Further, there could be data races introduced through an illegal transformation. This is a problem within the compiler and ARCHER does not pursue this class of errors.

2.6 Conclusions and Future Work

In this work we presented ARCHER, an OpenMP data race checker that embodies the design principles needed to cope with and exploit the characteristics of large HPC applications and their perennial development lifecyle. ARCHER seamlessly combines the best from static and dynamic techniques to deliver on these principles. Our evaluation results strongly suggest that ARCHER meets the design objectives by incurring low runtime overheads while offering very high accuracy and precision. Further, our interaction with scientists shows that it has already proven to be effective on highly elusive, real-world errors, which can significantly waste scientists' productivity.

However, our challenge does not end here. As part of bringing ARCHER to full production, we must further innovate. In particular, we need to reduce its runtime and memory overheads further so as to benefit a wide range of production uses. For this purpose, we will keep tapping into a great potential in the static analysis space. For example, ARCHER currently classifies each OpenMP region with the binary classification system: race free or potentially racy. More advanced technique will allow us to move away from the binary logic. In fact, we plan to crack open each of these potentially racy regions and apply fine-grained static techniques in order to identify and exclude race-free subregions within it. Exploiting symmetries in OpenMP's structured parallelism is another venue we plan to explore. Adequately defined symmetries will allow ARCHER to target a smaller set of representative threads and memory space for further overhead reduction.

Clearly, the aforesaid challenges cannot be pursued single-handedly. To enable communal participation (as noted earlier), we have released ARCHER in the public domain [26], and are looking forward to input from the community.

2.7 Summary

In this chapter we have presented the following contributions:

- The first practice and open-source data race checker for OpenMP programs.
- A precise, accurate, and scalable data race detection tools for real-world HPC applications.
- An extensive evaluation of ARCHER on a well-known benchmark and a HPC application.

- The successful result of ARCHER identifying, in a real-world OpenMP HPC application, data races that prevented the scientists to correctly port the program to the new supercomputer Sequoia at the LLNL.

CHAPTER 3

AN OPERATIONAL SEMANTIC BASIS FOR OPENMP RACE ANALYSIS

This chapter is based on work published on arXiv [55].

In the previous chapter we presented the OpenMP data race checker ARCHER whose technique, although experimental results showed its precision and accuracy, it suffers from high memory overhead. In this chapter we address some of the intrinsic limitations of ARCHER's technique through the definition of an operational semantics that formally defines the concurrency structure of OpenMP and enable a more precise and accurate data race detection analysis.

3.1 Introduction

OpenMP is the de facto standard for on-node parallelism in High Performance Computing. While OpenMP is highly portable and easy to use, it is also error-prone. Data races are one of the major source of errors in OpenMP based HPC applications. Although many existing correctness checking tools support programmers in the detection and removal of data races, these tools rely on static and dynamic analyses that either may not fit well with the needs of practical OpenMP race checking [10, 9, 56], miss races, or incur high overheads. Even symbolic analysis methods for OpenMP suffer from these issues [20].

Our past work has successfully adapted static and dynamic analysis to OpenMP and offered a practical race checker called ARCHER that has caught data races in critical field applications [23]. However, ARCHER suffers from high memory overheads, and misses races in many cases due to its exclusive reliance on the *happens before* model. It is well known that the races caught under this model depend on the schedule actually played out. That is, races within alternate schedules may be missed.

Our approach is to follow the lead of those who have exploited structured parallelism to make race-checking simpler and more efficient, for example for OpenMP [57], Cilk [58], and X10 [59]. We define an operational semantics that models the concurrency structure of OpenMP programs, exploiting the tool API (OMPT) [28] of modern OpenMP runtimes to identify every OpenMP event in the execution. Our approach also has the flavor of combining the exploitation of structured parallelism with lock-set based race checking (see Section 3.3.5 for our lock handling rules). The result is a more precise and traceable data race checker based on a clear operational semantics that fits in one page over 10 rules (Section 3.3.5), supported by some helper functions (Section 3.3.4). We believe that our formalization will benefit designers who seek to model new data race detection techniques for structured parallelism (in particular OpenMP) and those seeking to build and understand new and existing data race checkers.

Raman et al., in their work in this area [60, 27], propose techniques to exploit the structured parallelism on parallel programming models such as Cilk and X10, their techniques currently are focused on async/finish structured parallelism of X10 and Habanero-Java. This makes their technique not directly applicable to OpenMP at this point. To summarize, the main contributions of this work are:

- An operational semantics that model the concurrency structure of an OpenMP program matching the OMPT events, and an overview of a prototype race checker that demonstrates how such a semantics can be a workhorse for race checking.

- A set of rules that exploit the OpenMP structured parallelism to identify races.

- An extensible operational semantics that allows future OpenMP constructs to be captured and analyzed.

The remainder of this chapter is structured as follows: Section 3.2 discusses limitations of existing techniques that our operational semantics can overcome; Section 3.3 illustrates the state machine that implements the operational semantics rules, the conventions used to define the operational semantics rules, how we model the OpenMP constructs in our concurrency model, and a real example to show the effectiveness of the operational semantics in identifying data races; Section 3.4 gives some ideas of a possible implementation of the operational semantics in a real data race detection tool; Section 3.5 concludes the chapter.

3.2 Background

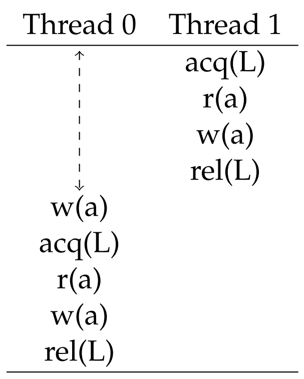
In this section we give an overview of the *happens-before relation* for dynamic data race detection analysis that underlies existing race detectors [23, 9, 14]. For our purposes, event a happens before [11] event b ($a \rightarrow b$) if (1) they occur in that order within the same thread, (2) if a is an unlock and b is a lock, or (3) they are synchronized otherwise (e.g., a is before a barrier and b happens after the barrier). A data race is a happens-before unordered pair of events where one event is a write. Vector-clocks [61, 62] and their adaptations [9] typically help realize happens-before. Happens-before is defined *per thread schedule*, thus making happens-before based race detectors miss races when they do not exercise all schedules. For example, in listing 3.1, we depict a parallel region with two threads. The main thread initializes a inside the master construct, and both threads write variable a within a critical section. Because the OpenMP master construct does not enforce an implicit barrier at its termination point, while thread 0 initializes a, the thread 1 can simultaneously access a within the critical section, introducing a data race.

Listing 3.1: Data race in OpenMP program that may not manifest at runtime.

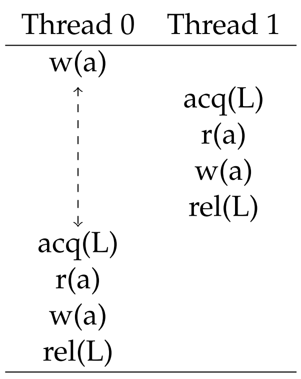
```
  int a;

#pragma omp parallel shared(a) num_threads(2)
{
#pragma omp master
  {
    a = 0;
  }
#pragma omp critical
  {
    a += 1;
  }
}
```

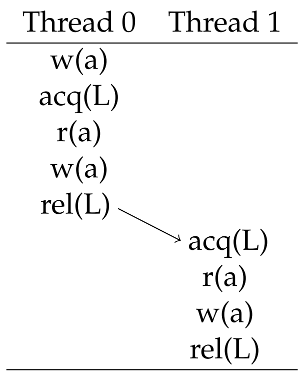
In Figure 3.1 we exhibit three different thread interleavings for the program in Listing 3.1. In the first two interleavings, the data race on a manifests itself. Indeed in Figure 3.1a, first thread 2 reads and writes within a synchronization block, while thread 1 performs a nonsynchronized write. As shown in the figure, the nonsynchronized write from thread 1 can happen anytime, even though thread 2 is accessing a within a critical section. In Figure 3.1b, first thread 1 performs a nonsynchronized write and thread 2 reads and writes within a synchronization block. In both cases, the two threads access simultaneously a and the data race detection algorithm shows the absence of happens-



(a) Interleaving 1, no happens-before (race detected).



(b) Interleaving 2, no happens-before (race detected).



(c) Interleaving 3, happens-before (no race detected).

Figure 3.1: Possible interleavings for program in Listings 3.1. The dashed line indicates the write operation of Thread 0 can happen simultaneously with the operations of Thread 1. The solid line indicates the happens-before edge between the threads.

before between the threads, catching the data race. On the other hand, in Figure 3.1c we have the typical situation where happens-before masks a race. Thread 1 executes both nonsynchronized and synchronized accesses on *a* before thread 2 performs any other operation. The release of the lock by thread 1 creates a happens-before edge with the acquiring of the same lock by thread 2, masking the previous nonsynchronized write by the first thread.

In our approach, races such as in Figure 3.1 are detected thanks to a global data structure that maintains relevant memory accesses information performed by the threads, along with other information such as operation type, thread id, and locks held while making accesses. At each barrier, the operational semantics verifies the presence of data races, analyzing all the memory accesses performed by the threads up to that point, ensuring no data race will be missed (details are in Section 3.3).

A key property of our operational semantics is that it highlights the concurrency structure created by a particular OpenMP program. If a particular thread forks two different threads and these threads perform their own accesses, our semantics records these accesses *not in terms of a particular interleaving*, but as a pair of accesses at *specific positions in the fork-join structure*, together with the *mutex locks held* when making the access. We exploit the idea of *offset span labels* pioneered by Mellor-Crummey [63] to record "positions" within the concurrency structure. We believe that these mechanisms serve the dual purpose of

(1) creating a concurrency representation that is general enough to "hang" on it future extensions to OpenMP's concurrency structure, and (2) also efficient enough to support the creation of a dynamic race detector.

3.3 Operational Semantics

The basic idea behind the operational semantics is to advance a state machine along the execution of the program in response to OpenMP events, and update the concurrency structure held in our state representation. Typical events include fork/join events (begin/end of a parallel region), acquiring and releasing of locks that guard critical sections, loads, stores, etc. The capturing of the OpenMP events is enabled by the new OpenMP Tools API (OMPT) [28] that modern OpenMP runtime implements to facilitate the development of correctness and performance tools. The OMPT interface triggers a callback for each OpenMP event that happens at runtime so that tools can access important information including parallel regions creation, threads entering or exiting a critical section, barrier executions, etc. The operational semantics rules match the OMPT events to correctly represent the concurrency structure of the OpenMP program. Each thread maintains a label in terms of offset-span labels that marks its *lineage* in the concurrency structure defined by prior forks and joins. Figure 3.2 illustrates the concurrency structure of the code in Listing 3.2, where circles represent the starting point of threads, and vertical lines represent traces of a thread's execution. Two or more diagonal lines that exit/enter the circles represent fork/join points in the program.

In our example, master thread 0 creates a parallel region of two threads (thread 1 and 2). Each thread creates a nested parallel region of a team of two. Because of the SIMD model followed, *all* threads in a parallel region execute the operations indicated at the horizontal tick marks.

Notice how each thread in the diagram has associated an id and a label which consists of pairs in square brackets. The id identifies the thread in the diagram, while the second label is the *offset-span label*. The offset-span label length grows at each fork and shrinks at each join.

When a thread reaches a fork, it creates a parallel region and a new pair of integers is added to the offset-span label. The first integer indicates the thread *rank* (ID) and the

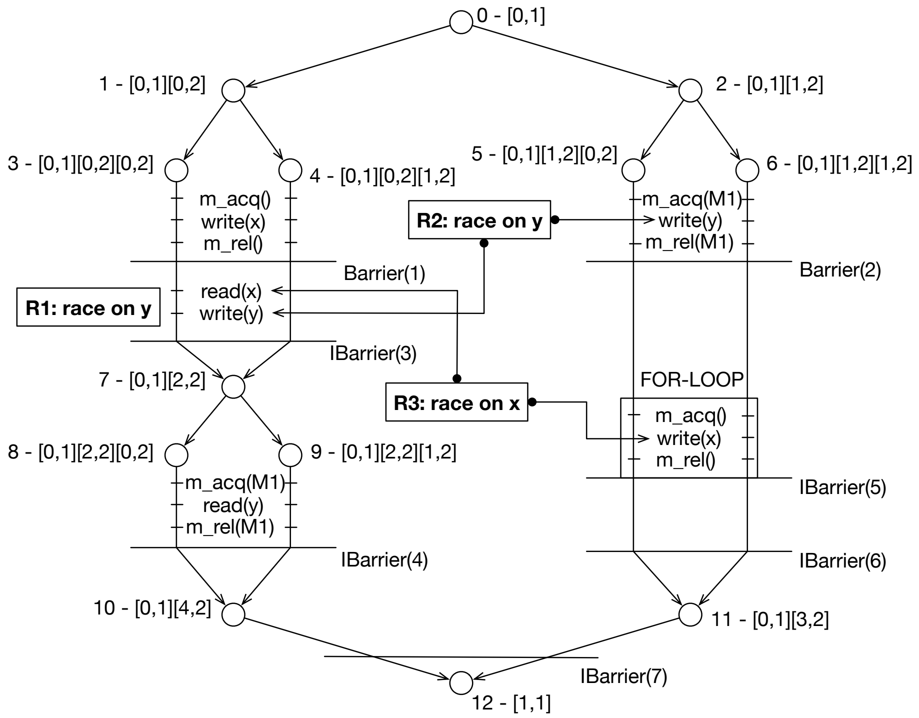


Figure 3.2: Structure of the OpenMP program in Listing 3.2.

second one indicates the number of the threads in the team. On the other hand, when threads join, the last pair of the label is removed and the previous label position is updated. We do not provide all the details of offset-span label manipulations here (see [63] for that); however, our semantic rules do include all the relevant details (Section 3.3.2). Mellor-Crummey has shown that given two threads and their offset-span labels, it is possible to determine if the two thread accesses are concurrent, and this happens to be the crux of race checking.

In our example of Figure 3.1, thread 0 creates the first parallel region and the operational semantics records this event through one of its rules. The same happens for thread 1 and 2 when they create the two nested parallel regions. At this point, each thread starts the execution of the operations in the program. In both nested parallel regions, the threads acquire *different locks* to access the shared variables. This triggers specific operational semantic rules to record the operations in the history of each thread.

More specifically, in the left parallel region, threads 3 and 4 enter a global critical section, write on x and exit from the critical section. At the same time, threads 5 and 6 in the nested parallel region on the right acquire a lock on $M1$, write on y and release

Listing 3.2: OpenMP program with nested parallel regions.

```
#pragma omp parallel shared(x, y) num_threads(2)
{
  if(omp_get_thread_num() % 2 == 0) {
  // Left-branch of the graph
#pragma omp parallel num_threads(2)
  {
#pragma omp critical
    {
      x = 1;
    }
#pragma omp barrier
    y = x;
  }
#pragma omp parallel num_threads(2)
  {
#pragma omp critical(N1)
    {
      printf("Y: %d\n", y);
    }
  }
  } else {
    //Right-branch of the graph
#pragma omp parallel num_threads(2)
    {
#pragma omp critical(N1)
      {
        y = y + 1;
      }
#pragma omp barrier
#pragma omp forg
      for(int i = 0; i < 10; i++) {
#pragma omp critical
        {
          x = x + 1;
        }
      }
    }
  }
}
```

the lock. Also, the loads and stores performed by threads trigger a rule that stores the information about the memory accesses in a global structure along with the thread id and the id of the mutexes previously acquired by the thread (if any).

In our example, threads 3 and 4 reach the barrier 1 eventually, while threads 5 and 6 reach barrier 2. When a parallel thread reaches a barrier (either implicit or explicit), it waits for all the other threads in the team; they then synchronize and proceed with the execution. The state machine triggers different rules at the barrier to model the thread synchronization–but more importantly *to perform the data race detection on the operations executed up to that point*.

The data race detection rule first identifies all possible concurrent threads in the system, comparing their offset-span labels. Second, it compares, for a given thread, its memory accesses with the memory accesses of another concurrent thread. If the rule identifies two memory accesses to a common location, at least one write, and without synchronization (or different mutex ids), it reports the race.[1]

Let us suppose the threads 8 and 9 have reached the implicit barrier 4, while the threads 5 and 6 are waiting at the implicit barrier 6. (Notice how threads 3 and 4 already joined into thread 7 which generated a new nested parallel region with threads 8 and 9. The global data structure still contains all the operations performed during the program execution up to those barriers.) All of the threads trigger the data race detection algorithm through one of the barrier rules. Up to that point, the global structure that collects the memory accesses contains all the loads and stores executed by the threads and related mutex information used for the memory accesses. The data race algorithm has all the information to identify potential data races. As stated previously, the algorithm identifies and compares only the memory accesses of concurrent threads.

In our example, there are three data races, identified by $R1$, $R2$, and $R3$.

- $R1$ happens within the same nested parallel region on shared variable y. This happens because both thread 3 and 4 (that are concurrent) write the shared location without any synchronization.

- Race $R2$ manifests between the threads of the two nested different parallel regions. The involved threads are 3 and 4 from the parallel region on the left, and 5 and 6 from the right parallel region. All the threads are concurrent to each other: threads 5 and 6 write on y through the critical section $M1$ and they do not race with each other. However, the concurrent threads 3 and 4 write on the same shared variable without any synchronization racing with threads 5 and 6.

[1]While these comparisons can make race-checking inefficient, our implementation in progress splits the burden into online event logging and offline event analysis that employs parallelism, as elaborated in Section 3.4.

- $R3$ is similar to thread $R2$ but on the shared data x.
- The data race detection algorithm identifies the races by comparing all the memory accesses in the global structure only for the possible concurrent threads. It is interesting to notice that the algorithm does not report any races on y between threads 3,4 and the threads 8,9. By comparing the offset-span labels, the algorithm recognizes that threads 3 and 4 have already terminated when threads 8 and 9 start their work, so they are not deemed concurrent.

We now detail our semantics, presenting each of its component building blocks in separate sections, followed by our semantic rules themselves.

3.3.1 Predicates and Conventions

We first need to state our conventions. $\mathbb{N}$ is the set of natural numbers, $\{0, 1, 2, \ldots\}$. $x \in \mathbb{N}$ can be treated as a set $\{0, \ldots, x-1\}$ as in set theory. Thus, $0 = \{\}$, $1 = \{0\}$, $2 = \{0, 1\}$, $3 = \{0, 1, 2\}$, etc. Whenever we treat a member of $\mathbb{N}$ as a number as well as a set, we'll make sure to provide a hint. $t \in TID$ is a thread identifier for some $TID \in \mathbb{N}$. $ADDR \in \mathbb{N}$ is the range of memory addresses accessed by the threads.

3.3.2 Offset-Span Labels

We showed how the *offset-span labels* are used to identify whether two threads are concurrent, and apply the data race detection only in that case. The offset-span label mechanism was introduced in [63]. An offset-span label, *osl* for short, labels each thread's execution point with a sequence of pairs, marking its lineage in the concurrency structure defined by prior forks and joins. The domain for the offset-span labels is $OSL = (\mathbb{N} \times \mathbb{N})^{\mathbb{N}}$, i.e. each member $osl \in OSL$ is a sequence of pairs:

$$[a_1, b_1][a_2, b_2], \ldots, [a_n, b_n].$$

Let us take two offset-span labels $osl_1, osl_2 \in OSL$, respectively associated to thread 1 and thread 2. These labels are sequential (hence the thread 1 and thread 2 are not concurrent) when:

case 1 $\exists_{P,S}(osl_1 = P) \wedge (osl_2 = PS)$, where P and S are any non-null sequence of ordered label pairs.

case 2 $\exists_{P,S_x,S_y,o_x,o_y,s}(osl_1 = P[o_x, s]S_x) \wedge (osl_2 = P[o_y, s]S_y) \wedge (o_x < o_y) \wedge (o_x \mod s = o_y \mod s)$ where P, S_x, and S_y are (possibly null) sequence of ordered pairs.

Otherwise, they are concurrent.

The offset-span label is an important piece of our concurrency model since it gives precious information regarding whether two given threads can actually race or not. For further details, please see [63].

3.3.3 System State

The state of the system consists of a global state GS and a set of thread local states TP (Thread Pool). The total state ts of any system is a pair "Global State, Thread Pool." A specific total state ts is:

$$ts = \langle gs, tp \rangle$$

Each total state ts originates from the domain TS, where $TS = GS \times TP$.
Each global state gs is a 4-tuple:

$$\langle bm, m, rw, \sigma \rangle$$

Each global state gs originates from the domain GS, where

$$GS = BM \times M \times RW \times \Sigma$$

where:

- The domain $BM = ParRegID \mapsto (\mathbb{N} \times \mathbb{N})$. Thus, for each $bm \in BM$, we have $bm : ParRegID \mapsto (\mathbb{N} \times \mathbb{N})$. Given a $p \in ParRegID$, bm returns a pair of natural numbers (a, b), where a is the "current Barrier Count" and b is the "target Barrier Count." When a thread t with offset-span label osl executes a $ParBegin(N)$ instruction, N threads are created, and an entry $\langle osl, (0, N) \rangle$ is added to function bm[2]. The first

[2]Recall that functions are single-valued relations, or sets of pairs with unique second component for each given first component. Thus, $\{\langle osl, (0, N) \rangle\}$ is a function. We allow functions to evolve, i.e. undefined for items explicitly added.

field a is incremented each time a thread hits a barrier. When the value reaches the number of threads in the team, it signals that all threads have synchronized at the barrier and the program can continue its execution.

- "Mutex" m comes from domain M where $M = Names \mapsto (\{-1\} \cup TID)$. That is, given a mutex name $m \in Names$, $M[m] = -1$ means that this mutex is free. Otherwise, $M[m] = t$, recording the fact that this mutex is held by the task associated to thread t. We use the value μ to indicate a mutex that has no name associated. A mutex with no name is usually the common case in a OpenMP progam and it refers to any global critical section or lock (e.g. `#pragma omp critical`).

- Let memory access-type $MAT = \{R, W\}$ indicates a read or a write operation of a memory access.

- $rw \in RW$ is a tuple (data structure) that maintains all the memory accesses of each thread in the system. We have $RW = TID \times OSL \times \mathbb{N} \times ADDR \times MAT \times M$. Each memory access performed by thread t is recorded as the tuple

$$\langle tid, osl, bl, addr, mat, mutex \rangle$$

  where:

  - $tid \in TID$ is the thread ID;
  - $osl \in OSL$ is the offset-span label;
  - $bl \in \mathbb{N}$ is the barrier label of the last barrier seen by the thread t;
  - $addr \in ADDR$ is the memory address;
  - $type \in \{R, W\}$ records reads or writes;
  - $mutex$ is the synchronization state (value of M in GS) at the time of the access;

- $\sigma \in \Sigma$ is the data state of the system, as described earlier.

The local state TP is the thread pool that contains a list of 3-tuples, each one of which is the local state of a thread:

$$\langle tid, osl, bl \rangle$$

The domain $TP = 2^{TID \times OSL \times \mathbb{N}}$ where:

- $t \in TID$ is the id of the thread;
- $osl \in OSL$ is an offset-span label;
- $bl \in \mathbb{N}$ is the label of the barrier the thread has witnessed last. We assume that each barrier instruction is of the form $bar(L)$ where $L \in \mathbb{N}$ carries the barrier number. A thread crossing a barrier sets its bl to the value L.

3.3.4 Helper Functions and Predicates

We define some helper functions to support the operational semantics rules. They can be operators or functions that receive some arguments in input and return a certain result or state useful for the rule execution. The helper functions are the following:

- *as*: is used as in Ocaml (it allows a name for a whole structure, as well as helps us refer to the inner details of the structure).
- $most(lst)$: we define *most* as a function that returns the same list given in input except the last element (i.e., in Python lst[:-1]).
- $\|$: This operator is used to describe that two different threads are concurrent. In particular, given two offset-span labels osl_1 for thread T_1 and osl_2 for thread T_2, $osl_1 \parallel osl_2$ (read osl_1 and osl_2 are concurrent) means that the threads T_1 and T_2 may race.
- $SpawnChildren(\langle ptid, posl, pbl \rangle, \sigma, N)$: Given the parent's thread id (*ptid*), offset-span label (*posl*) and barrier label (*pbl*), this function creates a pool of N threads — specifically, the local states of these threads $\langle tid, osl, bl \rangle$. It initializes the offset-span label *osl* for each thread created (e.g. at the beginning of a parallel region), by extending *posl* with pairs $[0, N]$ through $[N-1, N]$. The *bl* is set to *pbl*. The threads id are somehow uniquely generated.

- $GetChildJoin(tp)$: returns the single thread-state triple that result from fusing all the threads in the thread pool tp.

- $Concurrent(OSL, t_1, t_2)$ is the function that compares the offset-span labels as described in Section 3.3.2.

- $AddRW(\langle tid, osl, bl, addr, mat, m, n\rangle)$ adds the access into the *rw* structure. The record says "an access by *tid* with offset-span label *osl* and barrier label *bl* is performed at address *addr* with memory access type *mat*, when the mutex state is *m*."

- $Full(bm, osl)$: This predicate keeps the count of the number of threads that have reached a $ParEnd(N)$ (or a $Barrier(bid)$) construct. In order to count the threads, it uses the structure *bm* which is indexed by the $ParRegID$ represented by the offset-span label *osl*. In other word, the predicate *Full* means that other threads have reached the construct and have incremented the counter in the *bm* structure. From a functional language point of view *Full* would look like:

```
let Full(bm, osl) =
   let (count, N) = bm[osl]
   in (count == N - 1)
```

- $WaitAtBarrier(bid)$: This predicate is used for the example in Section 3.3.6 to indicate that a thread already encountered a barrier and it is waiting for the other threads in the team.

- $RaceFail(state, addr, t_1, t_2)$: This helper function is used to report the race found on *addr*, between thread t_1 and thread t_2.

3.3.5 Operational Semantics Rules

Now, we explain the rules in Figure 3.3 one by one. While each rule models a different behavior, all rules update the system state incrementing the program counter to point to the next instruction.

- Parallel Region Begin: The *ParallelBegin* rule models the creation of the team of threads for the encountered parallel region and initializes the offset-span labels for each thread.

$$\textbf{ParallelBegin}(\textbf{N})\frac{\begin{array}{c} at(tid,\sigma,ParBegin(N))\wedge \\ tp' = (tp - \{te\} \cup SpawnChildren(\langle tid,osl,bl\rangle,\sigma,N))\wedge \\ bm' = bm \cup \{\langle osl,(0,N)\rangle\} \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm',m,rw,\sigma'\rangle, tp'\rangle} \tag{3.1}$$

$$\textbf{ParallelEnd}(\textbf{N})\frac{\begin{array}{c} tp' \subseteq tp \wedge at(tid,\sigma,ParEnd(N))\wedge \\ \sigma' = nxt(\sigma,tid) \wedge tp'' = tp - tp' \cup GetChildJoin(tp')\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m,rw,\sigma'\rangle, tp''\rangle} \tag{3.2}$$

$$\textbf{ImplicitTaskBegin}()\frac{at(tid,\sigma,ImplicitTaskBegin()) \wedge \sigma' = nxt(\sigma,tid)}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m,rw,\sigma'\rangle, tp\rangle} \tag{3.3}$$

$$\textbf{ImplicitTaskEnd}()\frac{at(tid,\sigma,ImplicitTaskEnd()) \wedge \sigma' = nxt(\sigma,tid)}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m,rw,\sigma'\rangle, tp\rangle} \tag{3.4}$$

$$\textbf{LoadStore}()\frac{\begin{array}{c} at(tid,\sigma,LoadStore(addr,mat))\wedge \\ rw' = ADDR - RW(tid,osl,bl,addr,mat,mutex) \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m,rw',\sigma'\rangle, tp\rangle} \tag{3.5}$$

$$\textbf{AcquireMutex}(\textbf{name})\frac{\begin{array}{c} at(tid,\sigma,AcquireMutex(name)) \wedge m[name] = \emptyset\wedge \\ m' = m[name \rightarrow tid] \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m',rw,\sigma'\rangle, tp\rangle} \tag{3.6}$$

$$\textbf{ReleaseMutex}(\textbf{name})\frac{\begin{array}{c} at(tid,\sigma,ReleaseMutex(name)) \wedge m[name] = tid\wedge \\ m' = m[name \rightarrow \emptyset] \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm,m',rw,\sigma'\rangle, tp\rangle} \tag{3.7}$$

$$\textbf{Barrier}(\textbf{bid})\frac{\begin{array}{c} at(tid,\sigma,Barrier(bid)) \wedge Full(bm,most(osl))\wedge \\ bm' = bm - \{\langle osl,*\rangle\} \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm',m,rw,\sigma'\rangle, tp\rangle} \tag{3.8}$$

$$\textbf{Barrier}(\textbf{bid})\frac{\begin{array}{c} at(tid,\sigma,Barrier(bid)) \wedge \neg Full(bm,most(osl))\wedge \\ bm[most(osl)] \; as \; (count,N)\wedge \\ te' as(tid,osl,bid) \wedge tp' = tp - te \cup \{te'\}\wedge \\ bm' = bm \cup \{\langle osl,(count+1,N)\rangle\} \wedge \sigma' = nxt(\sigma,tid)\end{array}}{\langle gs,tp\rangle \longrightarrow \langle gs' \text{ as } \langle bm',m,rw,\sigma'\rangle, tp'\rangle} \tag{3.9}$$

$$\textbf{Barrier}(\textbf{bid})\frac{\begin{array}{c} te_1 \; as \; (tid_1,osl1,bl1) \in tp \wedge te_2 \; as \; (tid_2,osl2,bl2) \in tp \wedge (tid_1 \neq tid_2)\wedge \\ Concurrent(osl,tid_1,tid_2) \wedge i \in rw[tid_1] \wedge j \in rw[tid_2]\wedge \\ (rw[tid_1][i].addr == rw[tid_2][j].addr)\wedge \\ (rw[tid_1][i].mat == W) \wedge (rw[tid_2][j].mat == W)\wedge \\ (rw[tid_1][i].mutex \cap rw[tid_2][j].mutex = \emptyset)\wedge \\ (rw[tid_1][i].bl == rw[tid_2][j].bl) \wedge (rw[tid_1][i].bl \parallel rw[tid_2][j].bl)\end{array}}{\langle gs,tp\rangle \rightarrow RaceFail(\sigma,addr,tid_1,tid_2)} \tag{3.10}$$

Figure 3.3: OpenMP concurrency operational semantics

- Parallel Region End: The *ParallelEnd* rule models the end of the parallel region. It terminates the threads in the team except the master thread which resumes its execution.

- Implicit Task Begin: The *ImplicitTaskBegin* rule fires when a thread, after its creation, begins the associated implicit task which performs the work within the parallel region. This rule is a helper transition to initialize the thread and its implicit task state.

- Implicit Task End: The *ImplicitTaskEnd* fires when a thread exits the implicit barrier and the parallel region is terminating. It also resets the thread state.

- Load Store: The *LoadStore* rule triggers every time a thread performs a read or a write operation. Its task is to store the information about the current memory accesses of a thread along with other information such as the current locks held by the task, offset-span label, and so forth. The information about a load or a store are kept in a data structure shared among all threads.

- Acquire Mutex: The rule *AcquireMutex* fires when a thread encounters a synchronization construct, such as a critical section. It stores the id (μ in case of global critical section) of the synchronization construct into a data structure for the given thread. All the following memory accesses will be stored with the information that they happened within the given synchronization region.

- Release Mutex: The rule *Release Mutex* instead fires when a thread encounter the end of a critical section or release a lock. It removes, from the thread's data structure, the id of the synchronization construct.

- Barrier: The *Barrier* rules are of extreme importance since they implement the data race detection algorithm. The first two rules make sure that all threads in a team reached the barrier and update the information in the global state. Once all threads have hit the current barrier the third rule triggers and perform the race check. The data race check consists of searching for memory accesses conflicts between each given pair of concurrent threads. First, the rule checks if the pair contains two concurrent threads, either checking if they belong to the same barrier interval or comparing the offset-span labels. In the event the threads are concurrent, the rule applies the other checks to search for data races. It looks into the loads/stores data structures for memory accesses with the same address, checks if at least one of them is a write and they do not have any synchronization regions in common. In case all these checks are positive the rule triggers a *RaceFail* event to report the data race.

3.3.6 Operational Semantics Example

In this section we show an application of the operational semantics in an OpenMP example. We show how each rule is triggered according to the operations performed by the program. We also provide a transition table to illustrate the system state and how it changes under the execution of each rule. The example we use is the OpenMP program shown in Listing 3.1. Initially we have only the main thread, the total state of the system is therefore the following:

$$init = \langle gs, tp \rangle$$

with:

$$gs = \langle bm, m, rw, \sigma \rangle \in GS$$

$$tp = \langle tid, osl, bl \rangle \in TP$$

where:

$$gs = \langle \emptyset, \emptyset, \emptyset, \sigma \rangle$$

$$tp = \langle (0, [0,1], 0) \rangle$$

The Table 3.1 illustrates the transition table of the system for the example in Figure 3.4. Each thread in the table is represented by its thread id and offset-span label. The row 0 of the transition table shows the initial state of the system. The first fired rule is *ParBegin(2)* (Row 1) when the thread 0 hit the parallel construct. This rule models the beginning of the parallel region and the creation of the team of threads. In the example, the master thread creates one more thread to make a team of two. Both threads in the system trigger the *ImplicitTaskBegin* rule (Row 2 and 3) to initialize their status (e.g., offset-span labels, state, barrier counts, etc.). Now the threads start their parallel work. Thread 0 triggers the *LoadStore* rule (Row 4) when it accesses the master construct and initializes the variable *a*. The rule adds the memory access information inside the *rw* data structure and points to the next instruction. In the next instruction, thread 0 acquires the mutex which triggers

Table 3.1: State machine transitions for the example in Listing 3.1.

#	tid - osl	rule	bm		rw	tp	Next State
0	$Init$	—	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 0,[0,1],0\rangle$	$ParBegin(2)$
1	$0-[0,1]$	$ParBegin(2)$	$[0,1]=(0,2)$	$\emptyset$	$\emptyset$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	
2	$0-[0,1][0,2]$	$ImplicitTaskBegin()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$	$\emptyset$	$\emptyset$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	
3	$1-[0,1][1,2]$	$ImplicitTaskBegin()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\emptyset$	$\langle 1,[0,1][1,2],0\rangle$	
4	$0-[0,1][0,2]$	$LoadStore(x,W)$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	$AcquireMutex()$
5	—	$AcquireMutex()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	$LoadStore(x,W)$
6	—	$LoadStore(x,W)$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	$ReleaseMutex()$
7	—	$ReleaseMutex()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	$Barrier(1)$
8	—	$Barrier(1)$	$[0,1]=(0,2)$ $[0,1][0,2]=(1,2)$ $[0,1][1,2]=(2,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 0,[0,1][0,2],0\rangle$ $\langle 1,[0,1][1,2],0\rangle$	$WaitAtBarrier(1)$ $ImplicitTaskEnd()$
9	$1-[0,1][0,2][0,2]$	$AcquireMutex()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 1,[0,1][1,2],0\rangle$	$LoadStore(x,W)$
10	—	$LoadStore(x,W)$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$ $\langle 1,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 1,[0,1][1,2],0\rangle$	$ReleaseMutex()$
11	—	$ReleaseMutex()$	$[0,1]=(0,2)$ $[0,1][0,2]=(0,2)$ $[0,1][1,2]=(0,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$ $\langle 1,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 1,[0,1][1,2],0\rangle$	$Barrier(1)$
12	—	$Barrier(1)$	$[0,1]=(0,2)$ $[0,1][1,2]=(1,2)$ $[0,1][0,2]=(2,2)$	$\emptyset$	$\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\emptyset\rangle$ $\langle 0,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$ $\langle 1,[0,1][0,2],[0,1][0,2][0],x,W,\mu\rangle$	$\langle 1,[0,1][1,2],0\rangle$	$RaceFail(\sigma,x,0,1)$
13	—	$ImplicitTaskEnd()$	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 1,[0,1][1,2],0\rangle$	
14	0	$ImplicitTaskEnd()$	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 0,[0,1][0,2],0\rangle$	$ParEnd(2)$
15	—	$ParEnd(2)$	$\emptyset$	$\emptyset$	$\emptyset$	$\langle 0,[1,1],0\rangle$	

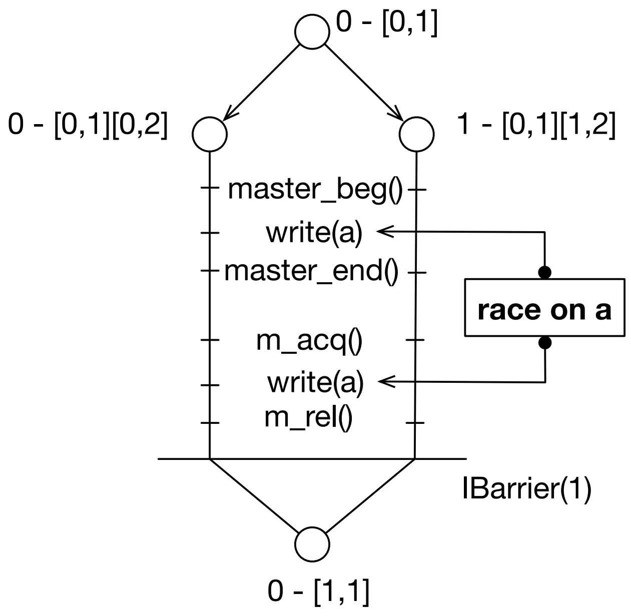


Figure 3.4: Structure of the OpenMP program in Listing 3.1.

the *AcquireMutex* rule (Row 5) and updates the thread state with the synchronization information. Thread 0 accesses again variable *a* and the *LoadStore* rule (Row 6) adds the new memory access to *rw* along with the synchronization information acquired by the previous operation. The thread 0 releases the mutex triggering the *ReleaseMutex* rule (Row 7) and reaches the implicit barrier at the end of the parallel region. The triggering of the *Barrier* rule (Row 7) keeps thread 0 on waiting for thread 1 to reach the barrier.

Thread 1 triggers respectively *AcquireMutex*, *LoadStore*, and *ReleaseMutex* (Row 9, 10, 11), which add a new synchronized memory access into the *rw* data structure. Now thread 1 reaches the implicit barrier triggering the *Barrier* rule (Row 12). The *Barrier* rule performs the data race detection which identifies the data race between the nonsynchronized access from thread 0 ($\langle 0, [0,1][0,2], [0,1][0,2][0], x, W, \emptyset \rangle$) and the synchronized access from thread 1 ($\langle 1, [0,1][0,2], [0,1][0,2][0], x, W, \mu \rangle$). The two accesses are performed by two different threads in the same memory location, both happen in the same barrier interval (concurrently according to offset-span label), at least one of the operation is a write, and one of them happens outside the critical section μ. The system reports the race through the *RaceFail* helper function.

The execution of the program continues triggering the *ImplicitTaskEnd* rule (Row 13 and 14) by both threads. Thread 1 terminates immediately, while thread 0 reaches the end of the parallel region and terminates with the end of the program.

3.3.7 Lowering OpenMP Constructs

Our operational semantics models the concurrency structure of an OpenMP program that uses a subset of the entire OpenMP specification [57]. We target OpenMP parallel directives and all related constructs except explicit tasks and target devices that we leave to future works. Our formalization lowers every OpenMP directive into basic underlying synchronization structures such as barriers and mutex. In the following paragraphs, we show how each of these directives can be simplified and modeled by the operational semantics.

- `parallel` **Construct**: The first five rules (3.1–3.4), in Figure 3.3, model the begin/end of a parallel construct including the creation and destruction of the implicit task associated to the threads. The threads within the parallel region trigger the other rules based on the work they are performing: accessing shared or private memory (3.5), acquiring/releasing mutexes (3.6,3.7), synchronizing to an implicit/explicit barrier (3.8–3.10). The data race detection algorithm performed at the barrier (either implicit or explicit) catches the potential race(s). The clauses related to the parallel region constructs do not influence the data race detection. For example, in presence of the *private* clause or similar, when the threads access their own private memory, the memory addresses of the locations are different for each thread, thus no race is reported.

- `worksharing` **Constructs**: The worksharing constructs such as *for*, *section*, *single*, and *workshare* are also supported by the operational semantics. These constructs add an implicit barrier at the end, so the race detection algorithm runs when the thread synchronizes, identifying any potential race within the barrier interval. In the presence of a *nowait* clause, the operational semantics models the specific constructs as an extension of the parallel work until the next barriers. Let us take the example in Listing 3.3. The snippet of code shows two consecutive parallel for-loops with the *nowait* clause. The clause removes the implicit barrier at the end of the first parallel loop, introducing a data dependency between the write on *a[i]* in the first loop and the read on *a[i]* and *a[i-1]* in the second loop. Consequently, all memory accesses performed by the threads in both loops happen in the same barrier interval. Only at the end of the second loop, when the threads encounter the implicit barrier, the state machine triggers the data race detection analysis (Rule 3.10). In detail, the state machine stores information about the memory locations accessed by the threads in both loops. Because of the data dependency between the loops, the race check identifies two common nonsynchronized memory accesses, in the *rw* data structure, from two different threads. Since one of the accesses is a write, the operational semantics reports the data race.

- `master` and `synchronization` **Constructs**: The only synchronization constructs not supported by the operational semantics are those related to tasking: *taskwait* and *taskgroup* which, as said previously, will be modeled in future work. When a thread encounters a synchronization directive, a rule logs the synchronization information for the current thread. Every memory access executed by the thread within a synchronization construct is collected in the *rw* data structure, with the information that the memory access are protected by a synchronization primitive. The data race detection, as shown in rule 3.10, uses this information to identify a potential nonsynchronized access and report the race.

Listing 3.3: Data race on array *a* because of *nowait* clause and data dependency between two for loops.

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i = 0; i < N; i++) {
    a[i] = 3.0 * i * (i + 1);;
  }
#pragma omp for
  for (i = 1; i < N; i++) {
    b[i] = a[i] - a[i - 1];
  }
}
```

3.4 Implementation

The operational semantics is a mathematical model and must clearly be adapted to real-world implementation settings. We have implemented a preliminary version of such a tool called SWORD. The main idea behind this tool is to log all OpenMP events and memory accesses into a file (one such file is created per thread). When the program execution terminates, an offline data race detection algorithm analyzes the log files to identify potential data races. The main advantages of this approach are: (1) dramatically reduced memory overheads compared to other tools (including ARCHER), and (2) parallelizable offline analysis.

More specifically, SWORD includes a compiler instrumentation pass for the source program and two checking phases. The compiler instrumentation inserts in the program, for each load and store, a call to a SWORD runtime routine that implements the event collection algorithm. Phase one consists of logging into files every memory access and synchronization operation that each thread executes at runtime. The SWORD runtime intercepts parallel regions begin/end, synchronization operations (e.g. critical sections, barriers, etc.), and other OpenMP events through the OMPT interface. This implementation benefits from our operational semantics directly including events that match OMPT events.

During the execution of the program, the SWORD runtime uses a buffer for each thread to collect the data regarding memory accesses and OpenMP events. When the buffer is full, SWORD compresses it, dumps it in a log file, and makes it available to collect new data. The use of data compression in this manner helps reduce memory overheads. Once the program finishes its execution, the log folder contains a log-file per thread.

The second phase consists of the offline analysis of the logs to identify the data races that manifested during the program execution. The algorithm identifies the pairs of concurrent threads using the offset-span label mechanism described in Section 3.3.2. The data race detection algorithm identifies memory conflicts between two concurrent threads. The algorithm obtains the information about the thread's memory accesses and synchronization operations from the logs, and looks for data races. Since the analysis requires only to read from the log files, the offline algorithm can be parallelized across multiple cores and a cluster of nodes to speedup the process.

3.5 Conclusions

In this work, we have presented an operational semantics to model the concurrency structure of OpenMP and enabling data race detection for structured parallelism. The operational semantics rules are straightforward and can serve as a valuable reference to everyday programmers. Also, the example 3.3.6 shows how our approach can identify data races even in corner cases where other techniques (e.g., those purely based on the happens-before tracking) can fail. In summary, our work provides a formalization to help researchers and tool developers to better understand OpenMP concurrency, and help them reliably and systematically build more precise data race checkers that reduce memory overheads.

As already described, we are working on a possible implementation of the operational semantics to support a new data race checker called SWORD. Details of the engineering of SWORD will be presented in future work.

To the best of our knowledge, our contribution is the first simple operational semantics to model the concurrency structure of OpenMP at a level that tool-builders care about. Our semantics is not yet suitable for those interested in issues such as (1) OpenMP's weak memory consistency model, (2) OpenMP's GPU offload features, and (3) OpenMP's tasking constructs. However, our semantics offers a very appealing starting point for such extensions.

The operational semantics rules mesh with the OMPT events providing a powerful as well as *standardized* instrumentation approach to represent the concurrency structure of an OpenMP program and enable targeted data race detection. We believe that with this formalization and the ongoing work we can build precise and accurate data race checkers that exploit the structured parallelism of parallel programming models such as OpenMP and its future incarnations.

3.6 Summary

In this chapter we have presented the following contributions:

- An operational semantics that model the concurrency structure of an OpenMP program matching the OMPT events.
- An overview of a prototype race checker that demonstrates how such a semantics can be a workhorse for race checking.
- A set of rules that exploit the OpenMP structured parallelism to identify races.
- An extensible operational semantics that allows future OpenMP constructs to be captured and analyzed.

CHAPTER 4

SWORD: A BOUNDED MEMORY-OVERHEAD DETECTOR OF OPENMP DATA RACES IN PRODUCTION RUNS

In the previous chapter we illustrated an operational semantics to formally define the concurrency structure of the OpenMP programming model and address the limitations of existing data race detection techniques. In this chapter we apply the operational semantics in practice with the implementation of a novel data race checker for large OpenMP applications, called SWORD. SWORD's goal is to serve as an instrument to identify data races in large OpenMP applications where existing techniques and tools fail.

4.1 Introduction

Given the inexorable march toward higher computational efficiencies, many critical software components are being transitioned to adopt on-node parallelism. The predominant parallel programming model of choice in this endeavor is OpenMP. Even though OpenMP provides constructs that ease the expression of parallelism, programmers still introduce egregious data races, resulting in corrupted answers that have seriously impacted critical projects [23, 25, 20]. Such accumulated evidence has raised awareness about the importance of race detection and elimination in high performance computing (HPC), and precipitated the creation of well-regarded data race benchmarks [64] that can help the community further tool development.

Currently, the mainstay for data race detection in HPC is dynamic analysis tools. Static-analysis-based data race detection tools, while known for their scalability, are unfortunately known also for their high false alarm rates, if they are to be reasonably complete [20, 56, 47, 8, 18]. Despite the practical successes of dynamic tools, there still is a paucity in terms of those that are effective on OpenMP applications.

There are currently four tools that can help with OpenMP race checking [64]: Helgrind [65], TSan [12, 66], Intel®Inspector XE [67], and ARCHER [23]. While Helgrind and TSan are well-engineered, mature tools, they are fundamentally designed for low-level models such as POSIX Threads. It is well-established that they generate false alarms when they do not recognize OpenMP synchronization semantics [64, 23]. Recently, ARCHER [23] introduced a technique to make an existing tool like TSan aware of OpenMP synchronization semantics, solving the false alarms issues.

The advantages of ARCHER in terms of coverage and the ability to handle large programs is presented in our previous work [23]. ARCHER is based on *happens-before race checking*, and owes its practical success to: (1) a static analysis phase that analyzes and excludes those statically guaranteed race-free loops from dynamic analysis, and (2) a well-engineered implementation of happens-before race checking provided by the TSan engine that employs *shadow memory* to log memory accesses. Experience shows that ARCHER can detect many data races in practice, and has helped find the root causes of show-stopper bugs in many mission-critical projects [23]. Despite these successes, all happens-before race checkers that employ shadow memory, including those used by ARCHER and TSan, suffer from three significant drawbacks:

4.1.1 Memory Overhead

These tools log read and write accesses, while assigning to them logical time instances (*e.g.*, vector clock values or epochs [9]). In such a setup, a data race can be missed unless *all* accesses are maintained. Unfortunately, this is practically impossible: there could be millions of program variables, with many being accessed millions of times. As a compromise, both TSan and hence ARCHER only maintain *four*[1] memory accesses per 8 bytes of application memory (hereafter called a *memory word*). Each access record (called a *shadow cell*) also occupies one word. Thus, it is clear that the memory consumption *quintuples* (and in practice, it goes up 6-fold due to other per-thread overhead). We have observed this when ARCHER was applied on the AMG2013 benchmark: the 6-fold increase with respect to total application memory gave us an out-of-memory (OOM) error. There is no easy way to predict application memory needs, and thus, OOM is a lurking danger *even* with only four shadow cells.

[1]A default setting, but adjustable between 1 and 8.

The main contribution of this chapter is a completely new race checker called SWORD *that runs in bounded memory*. Specifically, SWORD needs only a fixed 2 megabyte[2] per thread in auxiliary buffers to log traces. In addition, SWORD is exact, while keeping only four shadow cells is highly inexact—for instance, only four of the massive numbers of accesses per word are kept by TSan or ARCHER. All this is possible because SWORD does not employ happens-before race checking. Instead, each thread collects memory accesses into its own buffer. When this buffer fills up, the data is compressed and written out to disk. Then an *offline synchronization recovery and race analysis phase* detects races. This phase is driven by an operational semantics of OpenMP [55] that determines which accesses are concurrent.

4.1.2 Shadow-Cell Eviction

Clearly, with only four shadow cells, a fifth access to a memory word must evict one of the cells. Unfortunately, in HPC applications where the memory access intensity is high, this results in many missed races, as has been observed while using ARCHER on real-world applications. SWORD does not suffer from race omissions due to shadow-cell evictions since it relies on offline checking.

Our initial implementation of offline checking was inefficient for many examples. After careful optimization, we brought down its runtime from one day to a few seconds! More specifically, the techniques we use include:

- State-of-the-art self-balancing interval trees for recording and merging traces;
- An efficient realization of Offset-Span Labels [63] for concurrency discovery;
- Constraint solving to detect conflicting accesses through complex strided accesses and partial word overlaps.

Our trace collection uses OMPT, a tools interface that is expected to be incorporated into the future OpenMP standard [28]—thus facilitating portability. To limit the application slowdown, we collect traces from each thread/core in a completely uncoordinated fashion: that is, per-thread tracing does not wait for OpenMP barriers to finish. Instead,

[2]A user-adjustable bound, but we found that 2MB is typically optimal.

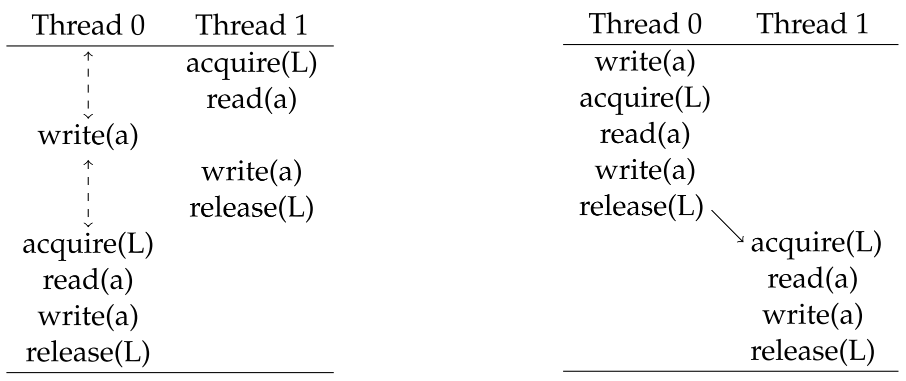
we comprehensively reconstruct synchronization information during our offline analysis phase, thanks to operational semantics logic that is directly *codified into the OMPT event interface,* and allows us to *recover all OpenMP concurrent regions exactly during offline analysis.*

4.1.3 Race Masking

A happens-before race checker can mask races when otherwise conflicting accesses are separated by a happens-before path created as an artifact of the particular schedule (see Figure 4.1). This form of race masking is reported in prior literature [68, 69]. We have also experienced ARCHER missing races in this manner. In this work, we instead introduce an accurate concurrency model on OpenMP's structural parallelism, thus avoiding such race masking. This again is a direct advantage accruing from our semantics. SWORD in fact guarantees completeness of data race checking under the absence of data-dependent control flows. *This guarantee cannot be provided by other OpenMP race checkers based on happens-before.*

To summarize, the contributions of SWORD are as follows:

- Bounded memory (about 3 MB) instead of taking gigabytes of shadow-cell storage.
- Free of race omissions due to shadow-cell evictions.
- No happens-before-induced race masking.
- Software available at `https://github.com/PRUNERS/sword`.



(a) No happens-before (race detected)

(b) Happens-before (no race detected)

Figure 4.1: Different interleavings generated by the same program. Dashed lines indicate that the write operations of Thread 0 can occur simultaneously with the operations of Thread 1. Solid lines indicate happens-before edges between the threads.

4.2 Background

A data race occurs when two concurrent memory accesses (one of which is a write) target the same memory location. Dynamic race detectors employ the happens-before relation (typically implemented using *vector clocks* [61] or variants) to determine whether two accesses are concurrent. The happens-before relation is a function of a thread schedule. Figure 4.1 shows two possible interleavings of the same program. In part (a), a race is caught because of the absence of any happens-before ordering between Thread 0's write(a) invocation and Thread 1's read(a) or write(a) invocation. In part (b), write(a) of Thread 0 is happens-before ordered before both read(a) and write(a) access of Thread 1, causing the race to be missed. This is one common source of missed races we observe in ARCHER. Notice that even without any branches in the code, the choice of interleavings decides whether a race is detected or missed. In SWORD, this sort of race omission does not happen, as the true concurrency status of two accesses is computed using an operational semantic model based on offset-span labels.

To further detail shadow-cell eviction mentioned in the previous section, consider the following example which harbors a race with respect to $a[0]$ because, while multiple threads read the array location $a[0]$, exactly one threads is arranged to write it without synchronization.

```
int a[N];

#pragma omp parallel for
for(int i = 0; i < N; i++) {
  a[i] = a[i] + a[0];
}
```

Suppose the master thread is the one writing $a[0]$ (assuming it got a head start). ARCHER may update the shadow cells related to $a[0]$ multiple times, and end up purging the access record of this write. This is because during the program execution, for each new memory access, the runtime updates one of the shadow cells randomly, overwriting the previous access information. Thus, when the other threads start the execution, the ARCHER runtime does not find any conflicting access on $a[0]$ (all 4 shadow cells hold read accesses) and therefore misses this race.

4.2.1 Operational Semantics for OpenMP Race Checking

Figure 4.2 shows the concurrency structure of an OpenMP program with two nested parallel regions, whose threads access shared memory locations. The figure depicts OpenMP barriers, as well as memory accesses and synchronization operations in between. In particular, a *barrier interval* is defined to be all the memory accesses and OpenMP operations that happen between two barriers. For example, barrier interval 3 includes the operations performed between barriers 1 and 3.

The concept of the barrier interval is important because the threads within the same barrier interval are concurrent and can race, but two threads separated by a barrier and belonging to two different barrier intervals cannot race. Whenever a thread-specific access event or synchronization event is fed to our operational semantic model, its *structural-operational-semantics-based* state representation always keeps an updated notion of which barrier interval a thread is currently living within. For example, the data race *R1* happens within the same barrier interval 3 between thread 3 and 4, since they both write on y with no synchronization. On the other hand, there is no race between the write on x in barrier interval 1 by thread 3 and the read on x by thread 4 in barrier interval 3 because those accesses are indeed separated by a barrier.

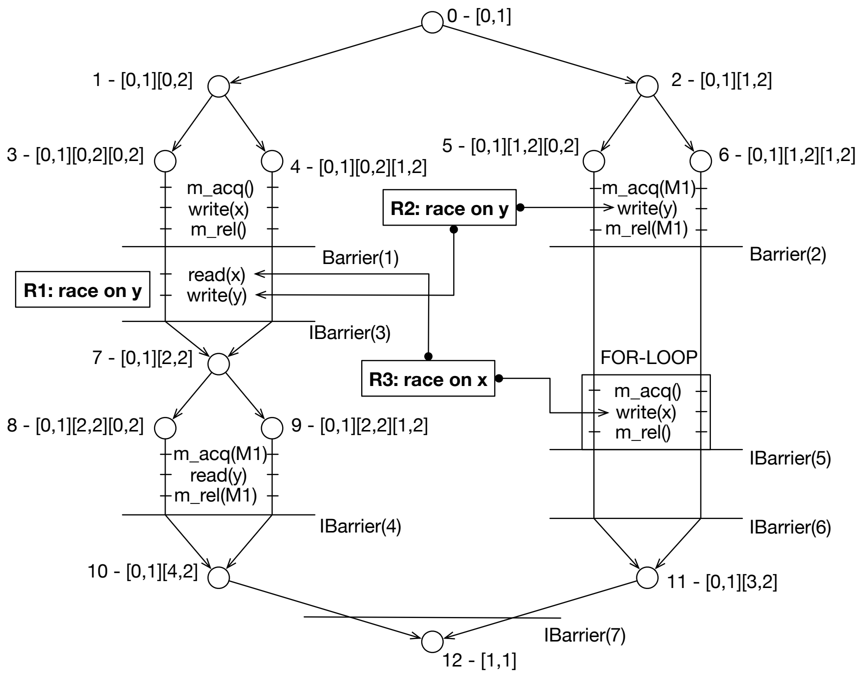


Figure 4.2: Structure of an OpenMP program

In the case of nested parallelism, two threads that belong to two different barrier intervals can in fact race—for example data races *R2* and *R3*. These two data races happen because the threads are accessing shared variables (y for R2 and x for R3) from two barrier intervals that belong to different concurrent parallel regions. Our operational semantics relies on *offset-span labels* to identify if two threads can race, so that we can apply the data race analysis only to concurrent threads.

4.2.2 Offset-Span Labels

An offset-span label tags each thread's execution point with a sequence of pairs (e.g., $[0,1][0,2][0,2]$), marking its lineage in the concurrency structure defined by prior forks and joins. By comparing these labels we can determine if two threads are concurrent, thereby focusing the data race analysis only to potentially racy threads.

The domain for the offset-span labels is $OSL = (\mathbb{N} \times \mathbb{N})^{\mathbb{N}}$, i.e., each member $osl \in OSL$ is a sequence of pairs $[a_1, b_1][a_2, b_2], \ldots, [a_n, b_n]$. A pair consists of *offset* and *span*. The span indicates the number of threads spawned by the fork (e.g., begin of parallel region) from which the pair is descended. The offset distinguishes the pair among the other pairs descended from the same parent. For example, let us take the label $[0,1][0,2][0,2]$ from thread 3 in Figure 4.2. Starting from the end, the pair $[0,2]$ indicates that the thread has id 0 in a parallel region of two threads; the second pair $[0,2]$ is the thread's parent with id 0 in a parallel region of two threads; the first pair $[0,1]$ is the predecessor of the thread's parent and represents the master thread.

Let us take two offset-span labels $osl_1, osl_2 \in OSL$ associated to thread 1 and thread 2, respectively. These labels are sequential (i.e., thread 1 and thread 2 are not concurrent) when:

case 1 $\exists\ P, S\ .\ osl_1 = P \wedge osl_2 = PS$, where P and S are non-empty sequences of pairs.

case 2 $\exists\ P, S_x, S_y, o_x, o_y, s\ .\ osl_1 = P[o_x, s]S_x \wedge osl_2 = P[o_y, s]S_y \wedge o_x < o_y \wedge o_x \bmod s = o_y \bmod s$, where P, S_x, S_y are (possibly empty) sequences of pairs.

Otherwise, the labels are concurrent.

SWORD obtains its completeness because (1) it collects and analyzes *every* memory access, and (2) our operational semantics is faithfully followed by our implementation. For instance, the happens-before based technique can miss data races based on the mutex contention order (Figure 4.1(b)). SWORD's approach is independent of mutex contention *so long as* downstream control-flows are not affected by the mutex acquisition order.

4.3 SWORD Technique and Implementation

4.3.1 Dynamic Analysis

4.3.1.1 Compiler Instrumentation

We implemented SWORD using the LLVM/Clang tool infrastructure [36] (see Figure 4.3). Our LLVM instrumentation pass instruments all load and store instructions that are executed within a parallel region. (We ignore sequential instructions as they cannot race.)

4.3.1.2 Log Collection

At runtime, SWORD collects all the information necessary for the offline data race detection. The collection of logs is fully parallel: every thread concurrently gathers information regarding its own memory accesses and OpenMP events. SWORD interacts with the OpenMP runtime through the OMPT interface to gather all the information regarding thread creation, parallel region begin/end, and synchronizations points (e.g., barriers, critical section). Meanwhile, the instrumented parallel loads and stores gather information about every parallel memory access (e.g., size, read or write, atomic). Each thread maintains one log file and one metadata file. The log file contains the information about memory accesses and OpenMP events, while the metadata file contains the IDs of parallel regions, offsets into the log file to obtain the data (i.e., memory accesses and OpenMP events) regarding a specific parallel region, and other information. Table 4.1 details each thread's metadata file, which helps the offline analysis identify the concurrency structure. Each line in the metadata file represents a barrier interval. This information is used by the offline data race detection algorithm to extract from the log file the chunk of data for a specific barrier interval.

During the program execution, SWORD collects the memory accesses and OpenMP

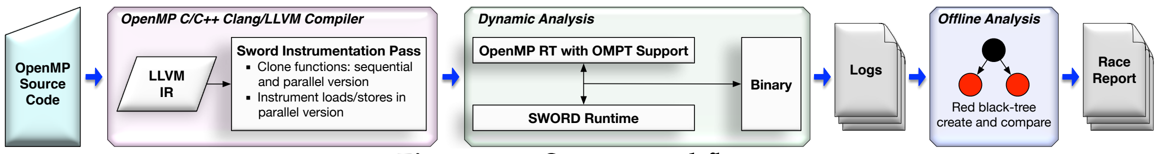

OpenMP Source Code
OpenMP C/C++ Clang/LLVM Compiler
LLVM IR
Sword Instrumentation Pass
▪ Clone functions: sequential and parallel version
▪ Instrument loads/stores in parallel version
Dynamic Analysis
OpenMP RT with OMPT Support
SWORD Runtime
Binary
Logs
Offline Analysis
Red black-tree create and compare
Race Report

Figure 4.3: SWORD tool flow

Table 4.1: Example of thread's metadata file. Each line corresponds to one barrier interval. Column **pid** is parallel region ID, **ppid** is parent parallel region ID, **bid** is barrier ID, **offset** and **span** define offset-span label, **level** is level of parallelism, **data begin** is offset (in bytes) in the log file of the beginning of the respective data chunk, **size** is its size.

pid	ppid	bid	offset	span	level	data begin	size
0	–	0	0	24	1	0	50,000
0	–	1	0	24	1	50,000	75,000
1	–	0	0	24	1	75,000	10,000

event information into limited-size thread-local storage buffers. Once the buffer is full, it is compressed and asynchronously written out into log files. We compared several open-source compression algorithms, namely *LZO* [70], *Snappy* [71], and *LZ4* [72]. In our case, they all have similar performance and compression ratios, and we chose *LZO* since it was easier to integrate into SWORD.

4.3.1.3 Bounded Dynamic Analysis Overhead

As previously mentioned, during the dynamic analysis each thread maintains a thread-local storage buffer to collect memory accesses and OpenMP events before writing them into a file. We fine-tuned the buffer size to minimize cache misses, and we found that an optimal size for our setup holds 25,000 events, amounting to around 2 MB total. The SWORD runtime also maintains other information in several thread-local storage variables. The amount of memory needed by SWORD for all these auxiliary buffers and OMPT is about 1.3 MB per thread. Given that the memory overhead is bounded and independent of the characteristics of the analyzed application, we define a formula representing the total memory overhead of SWORD. Let N be the number of threads, B the memory overhead introduced by SWORD per thread, and C the memory overhead introduced by the OMPT interface. Then, the total memory overhead of SWORD is $N \times (B + C)$. Our experimental results show that in our setup the total memory overhead of SWORD is around 3.3 MB per thread.

4.3.2 Offline Analysis

Offline analysis starts by analyzing the metadata files to identify the concurrency structure. Once the algorithm has identified all pairs of concurrent barrier intervals and threads, it obtains information about the memory accesses and OpenMP synchronization operations from the log files. The metadata file contains an offset for each barrier interval indicating the location of pertinent data in the log files. The size of a single log file can be *dozens of gigabytes*, and hence the entire data collection from an application can be of the order of terabytes. Thus, even without application memory pressure, it is not always possible to analyze all the data directly in memory. To handle large log files efficiently, we employ a *streaming algorithm* [73] approach that reads access information from log files in small chunks and carries out our analysis.

For each thread, the algorithm builds an *interval tree* to summarize memory accesses and to maintain information about OpenMP events. In our implementation, we use an augmented red-black tree [74] to maintain the interval tree balance and to speed up the operations of insertion and search. A node in an interval tree contains the range of memory accesses[3] it represents, and auxiliary information such as the operation type (R/W), size of the access, stride of the interval, program counter, and mutex set. The interval tree approach allows us to summarize the information about consecutive memory accesses (e.g., array accesses) in one node. The data race detection is performed by comparing the interval tree of each thread to the interval trees of other concurrent threads. When a node in the tree overlaps with a node of another tree there is a *potential race*.

Figure 4.4 shows an example of two threads accessing an array of structures. Each thread is accessing a different field of the structure, performing either a read or write, and there is not overlap in the accesses—hence no data race. During the offline analysis, SWORD summarizes the accesses of both threads using the two shown intervals. The two intervals do overlap; *however, if we consider the size and the stride of the accesses, they do not actually have any addresses in common*, as the threads are accessing different memory addresses. Thus a simple overlap check is not sufficient to identify whether two intervals intersects.

In our offline race detection algorithm, we use all the available interval information (e.g., count, stride) to check if two intervals have memory addresses in common. For an interval of thread T_i, we represent all addresses that belong to it with the following constraint:

[3]We treat a single access as a range with the same beginning and end.

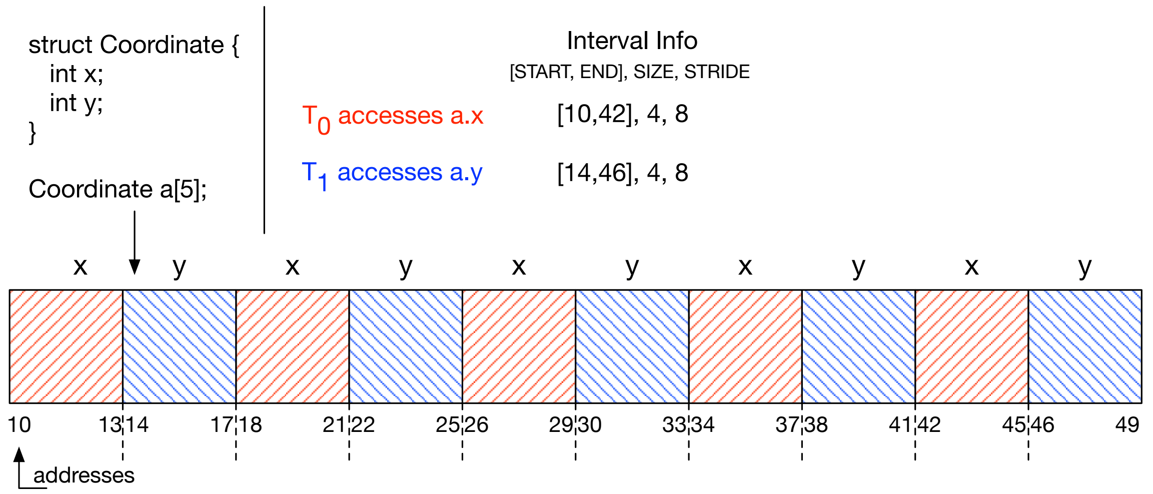


Figure 4.4: Example of threads that access the same memory interval but do not have common addresses

$$
\begin{aligned}
&\Delta \cdot x_i + b_i + s_i = a \\
&\wedge\ 0 \leq x_i \leq ((b - e)/\Delta) + 1 \\
&\wedge\ 0 \leq s_i < s,
\end{aligned}
$$

where a is an address belonging to the interval, b and e are the starting and ending address of the interval respectively, Δ is the stride, and s is the size of the memory access. If we consider the example of Figure 4.4, we can represent all the addresses for intervals of T_0 and T_1 with these constraints:

$$
\begin{aligned}
T_0 :\ & 8 \cdot x_0 + 10 + s_0 = a \\
& \wedge\ 0 \leq x_0 \leq 5 \\
& \wedge\ 0 \leq s_0 < 4
\end{aligned}
\qquad\qquad
\begin{aligned}
T_1 :\ & 8 \cdot x_1 + 14 + s_1 = a \\
& \wedge\ 0 \leq x_1 \leq 5 \\
& \wedge\ 0 \leq s_1 < 4
\end{aligned}
$$

If their conjunction is satisfiable, then the threads are accessing a common address. Furthermore, if at least one of the operations is a write, then a race is reported. In our implementation, we use integer linear programming to solve the constraints, and in particular GNU GLPK Version 3.63 (any other solver with similar capabilities could be employed).

The algorithm complexity is $O(Nlog(N))$ for the interval tree creation with N being the number of memory accesses: it takes $O(log(N))$ to insert a node into a tree and this is done for all N memory accesses. The comparison of two interval trees is $O(Mlog(M))$ with M being the number of nodes in the tree: each of the M nodes in a tree is compared to the other trees, which is a binary search with complexity $O(log(M))$. Note that $M \leq N$

since the interval tree can summarize multiple access into one interval node.

4.3.2.1 Interval Tree Example

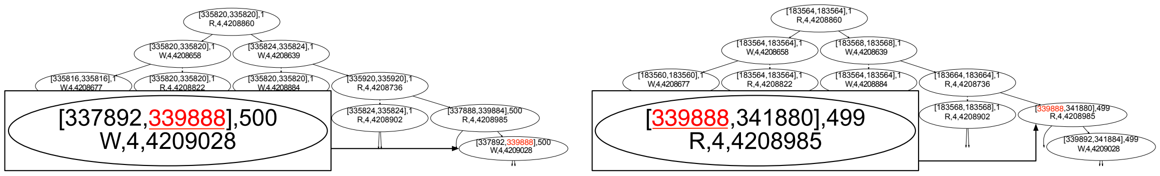
The following example, when executed with two threads, contains a data race in the array *a* due to a data dependency:

```
int a[1000];

#pragma omp parallel for num_threads(2)
for(int i = 1; i < 1000; i++) {
  a[i] = a[i - 1];
}
```

During the dynamic analysis, SWORD generates two log files and two metadata files. Since the program has only one parallel region and one barrier interval, the metadata files contain only one line. The offline data race detection algorithm extracts the barrier interval data using the metadata files, and builds one red-black interval tree per thread.

Figure 4.5 shows the possible interval trees for the two threads executed by the program. Each node in the interval tree describes a memory access or a collection of memory accesses (e.g., array access). In addition, each node has fields to hold information about the type of operation (read or write), size of the memory access, program counter, and list of mutexes held for that specific memory access. When the algorithm identifies two overlapping intervals, as shown in red/underlined in Figure 4.5, it uses the additional information in nodes to construct the integer linear constraint to check if there is a potential race. The algorithm also checks whether one of the intervals is a write operation and if the intersection of the mutex lists are empty. If these two condition are met and the linear constraint is feasible, a race is reported. In the case of Figure 4.5, the two red/underlined



(a) Interval tree for Thread 0 (b) Interval tree for Thread 1

Figure 4.5: Example interval trees. The red/underlined nodes are the two overlapping intervals that identify the race. The node's fields represent, respectively, begin, end of the interval, count, type of operation, access size, and program counter.

intervals are overlapping since they have an address in common. Therefore, SWORD reports a race at the lines of code associated to the program counter stored by the intervals.

4.3.3 Limitations

Although SWORD supports most of the constructs defined by the OpenMP specification, in its current form it cannot analyze programs based on OpenMP tasking. The main limitation for supporting OpenMP tasking is that the current formulation of the offset-span label mechanism does not allow to identify whether two threads that executed two different tasks are concurrent or not. This is critical to avoid false alarms and missed races. Despite this limitation, programs that employ OpenMP tasking are still rare, thus SWORD can analyze most of the existing OpenMP applications.

4.4 Experimental Results

We evaluate SWORD on two OpenMP microbenchmark suites and four large real-world HPC applications. More specifically, we select DataRaceBench [64] and OmpSCR [39] OpenMP microbenchmarks to show the effectiveness of SWORD in terms of identifying data races. In addition, we use real-world HPC applications to assess its performance and memory overhead. We compare SWORD against the state-of-the-art OpenMP data race checker ARCHER [23].[4] In our experiments, we run two configurations of ARCHER: with default settings and with the "flush shadow" option enabled. The purpose of enabling this option, which flushes memory between independent parallel regions, is to try to reduce the memory overhead of ARCHER and to have a more fair comparison with SWORD. We also use the default setup of 4 shadow cells per 'line' (see Section 4.2).

We perform our evaluation on a machine with two 12-core Intel Xeon E5-2695v2 processors, 32GB of RAM, and 800GB of SSD storage. The machine runs the TOSS Linux distribution (kernel version 3.10), which is a customized distribution specifically optimized for HPC clusters. We average the measured runtimes and memory overhead of all benchmarks across 10 executions, and we vary the number of threads from 8 to 24. In the experimental results, "baseline" denotes the original benchmark characteristics with data

[4]We also performed a preliminary comparison with the latest version of Intel®Inspector XE. We obtained results that are very similar to its comparison with ARCHER from our previous work [23]. Hence, we omit a detailed comparison with Intel®Inspector XE from this work.

race checking disabled, while "archer" and "archer-low" denote ARCHER in its default and low memory overhead configuration respectively, and "sword" denotes our SWORD tool.

4.4.1 DataRaceBench Microbenchmarks

The DataRaceBench microbenchmark suite [64] consists of small OpenMP codes with and without data races; each 'racy' benchmark contains one known data race documented by the authors. We run every tool on all benchmarks and inspect the outcomes; none of the tools report false alarms, and they also successfully identified almost all races. All tools missed the races in benchmarks `indirectaccess{1-4}-orig-yes`. These data races do not manifest along all program paths, and given that both SWORD and ARCHER are dynamic analysis tools that analyze only the executed control flow, they can miss such races. In benchmarks `nowait-orig-yes` and `privatemissing-orig-yes`, SWORD analysis is more complete and it reports races that ARCHER misses for the reasons discussed in Section 4.2. These are all *read-write* data races happening in the same shared variable and parallel region. Because of multiple reads by the same thread, the shadow cells maintained by ARCHER are eventually overwritten, and this information loss causes these races to be missed. SWORD does not suffer from such information loss, and it correctly identifies them. Note that all tools report an additional unknown race in `plusplus-orig-yes`, and SWORD reports an additional unknown race in `privatemissing-orig-yes` as well. These are not false alarms, but rather real races that the authors of the benchmarks failed to document; they will fix this in the next release. Finally, since DataRaceBench benchmarks are small, the runtime and memory overheads are similar among the tools.

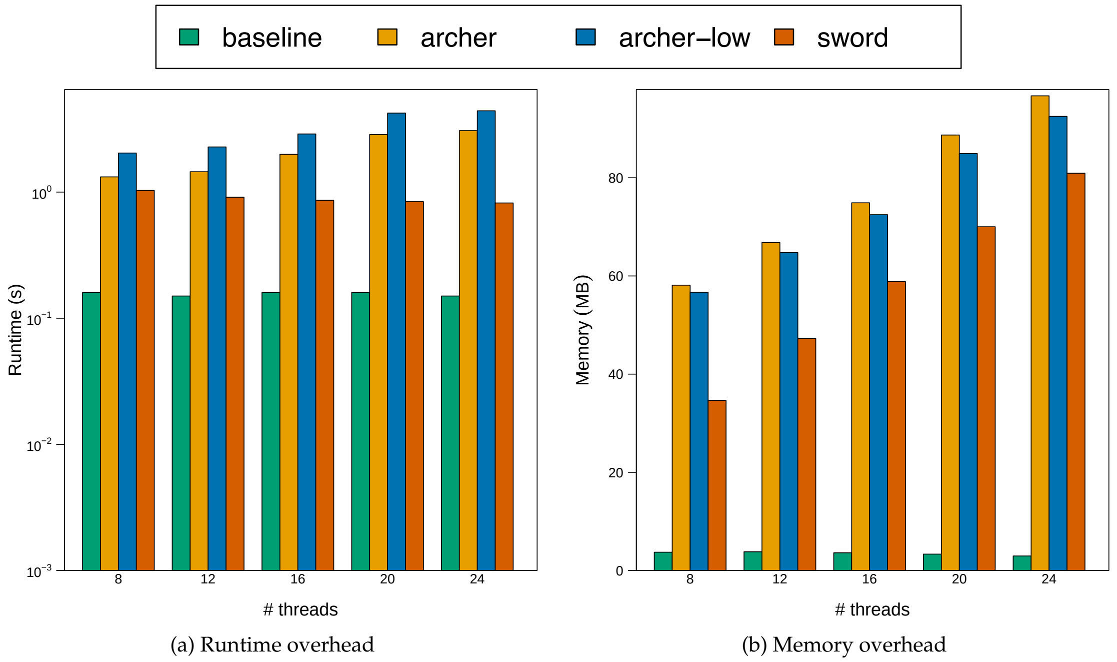
4.4.2 OmpSCR Microbenchmarks

The OmpSCR benchmark suite contains known data races that have been documented in previous works [39, 23]. Table 4.2 gives the number of data races detected by each tool. (We again omit race-free benchmarks since we verified that none of the tools report false alarms.) SWORD not only identifies the same races as ARCHER, but also detects new undocumented races in the following benchmarks: `c_md`, `c_testPath`, `cpp_qsomp1`, `cpp_qsomp2`, `cpp_qsomp5`, and `cpp_qsomp6`. Our manual inspection confirmed that all these races are real. ARCHER misses these races for the reasons discussed in the previous sections.

Table 4.2: Data races reported in OmpSCR suite

	# of Reported Data Races		
Benchmark	**archer**	**archer-low**	**sword**
c_loopA.badSolution	1	1	1
c_loopB.badSolution1	1	1	1
c_loopB.badSolution2	1	1	1
c_md	1	1	2
c_testPath	2	2	6
cpp_qsomp1	1	1	2
cpp_qsomp2	1	1	2
cpp_qsomp5	1	1	3
cpp_qsomp6	1	1	2

Figure 4.6 gives the geometric mean of the runtime and memory overheads to indicate the overall tendency of the values, considering the large gaps in execution time and memory usage among the different benchmarks. The runtime overhead is small for all tools, while the relative memory overhead is large due to small baseline, but still less than 100 MB for all tools. Also note that the memory overhead of SWORD is constantly around 3.3 MB per thread, as we indicated in Section 4.3. When compared, the runtime and memory overhead of the SWORD data collection is lower than ARCHER in both configurations. The plots do not include the runtime and memory overhead of the offline data race detection



(a) Runtime overhead (b) Memory overhead

Figure 4.6: Geometric mean of runtime and memory overhead for OmpSCR suite; the number of threads varies from 8 to 24.

algorithm, which may increase the total amount of resources needed by SWORD for a complete analysis.

Table 4.3 shows the overheads of the offline data race checking with SWORD compared to the two ARCHER configurations. The runtime overhead depends on the size of log files and the number of parallel regions the algorithm has to analyze for each benchmark. We parallelized the offline analysis across a cluster of nodes and the results show that the offline data race detection can last from a few milliseconds up to a few seconds. However running the entire offline analysis sequentially for some of the benchmarks can take several minutes. We omit the memory overhead for the dynamic analysis because it is negligible given the small size of the benchmarks. While for most of the benchmarks the dynamic

Table 4.3: Overheads on the OmpSCR suite executed with 24 threads, including the execution time of the parallel offline analysis. Column **baseline** is the baseline runtime; **archer** is the ARCHER runtime; **archer-low** is the low memory overhead ARCHER configuration runtime; **DA** is the total dynamic analysis runtime including logging; **OA** is the offline analysis runtime when executed sequentially; **MT** (Max Time) is the longest offline analysis runtime for any parallel region indicating how long parallel analysis runs; **#PR** is the number of independent parallel regions to analyze; **LS** is the amount of storage required to store the generated log files.

Benchmark	**baseline(s)**	**archer(s)**	**archer-low(s)**	**sword**				
				DA(s)	**OA(s)**	**MT(s)**	**#PR**	**LS**
c_fft	0.13	0.81	0.84	0.52	2.09	1.34	2	2.4MB
c_fft6	0.03	0.14	0.15	0.12	0.12	0.12	1	122kB
c_jacobi01	0.9	19.83	20.91	2.57	2.06	1.33	2	51MB
c_jacobi02	0.89	19.64	20.38	2.59	0.63	0.63	1	51MB
c_loopA.badSolution	0.03	0.47	1.59	0.18	3.16	0.35	100	394kB
c_loopA.solution1	0.03	0.65	2.76	0.36	5.88	0.22	200	981kB
c_loopA.solution2	0.03	0.3	0.39	0.27	0.14	0.14	1	452kB
c_loopA.solution3	0.03	0.3	1.43	0.23	2.33	0.17	100	458kB
c_loopB.badSolution1	0.03	0.47	1.62	0.3	3.03	0.14	100	398kB
c_loopB.badSolution2	1.79	4.08	5.26	2.26	3.09	0.15	100	390kB
c_loopB.pipelineSolution	0.03	0.28	0.32	0.25	0.14	0.14	1	462kB
c_lu	0.04	10.5	15.81	0.83	25.35	0.28	499	20MB
c_mandel	0.08	5.06	5.05	0.37	0.1	0.1	1	81kB
c_md	0.47	80.87	84.47	3.65	0.55	0.17	21	1.5MB
c_pi	0.02	0.14	0.17	0.14	0.11	0.11	1	81kB
c_qsort	0.04	0.23	0.33	0.14	0.27	0.12	10	125kB
c_testPath	0.03	0.26	0.33	0.26	0.09	0.09	1	81kB
cpp_qsomp1	1.38	259.9	264.32	5.46	1.76	1.76	1	321MB
cpp_qsomp2	1.38	262.8	263.19	5.39	1.82	1.82	1	303MB
cpp_qsomp5	14.27	41.54	41.51	55.44	16.47	16.47	1	204MB
cpp_qsomp6	1.52	263.51	263.16	5.36	1.93	1.93	1	316MB
Mean	1.1	46.28	47.33	4.13	–	–	–	–
Median	0.04	0.81	2.76	0.37	–	–	–	–
Geometric Mean	0.15	0.81	2.76	0.37	–	–	–	–

analysis terminates quickly and does not differ much from ARCHER runtime overhead, for some of the applications the offline analysis can take a considerably long time.

4.4.3 HPC Benchmarks

We assess the performance and memory overhead of SWORD using four small to large-size HPC benchmark codes. We use three codes, namely AMG2013, LULESH, and miniFE, from the CORAL benchmark suite [40], while the fourth code HPCCG is a part of the Mantevo project [75]. These codes model scientific problems and simulations, and their size ranges from tens to hundreds of thousands of lines of code. We also leverage AMG2013 to evaluate the overheads of the tools with an increasing problem size. AMG2013 is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. Therefore, we perform the evaluation using 4 different grid sizes: 10^3 (AMG2013_10), 20^3 (AMG2013_20), 30^3 (AMG2013_30), and 40^3 (AMG2013_40).

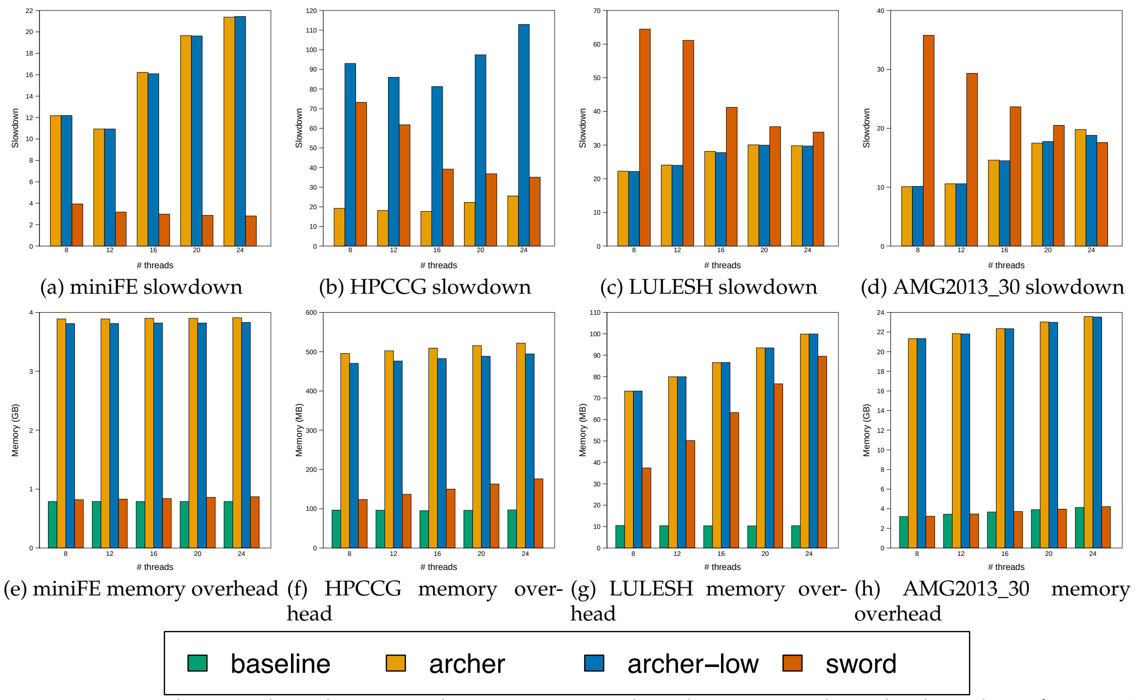
Table 4.4 shows the number of data races detected by each tool. Note that none of the tools report false alarms. Both tools find one race in HPCCG, which happens in a parallel region where all threads are writing the same value into a shared variable. While this race may seem harmless, it in fact results in undefined behavior based on the C/C++ standard, and compiler optimizations could unpredictably modify the outcome of this program [23, 32]. ARCHER detects 4 known races in smaller-scale AMG2013 runs [23], while it runs out of memory at large scale. SWORD both completes the analysis at large scale and detects 10 additional races missed by ARCHER. These races happen in the same large parallel region (around 400 LOC) as the others, and they are all the same type of read-write races. As before, ARCHER misses them since it maintains only a limited number of previous

Table 4.4: Data races reported in HPC benchmarks. OOM indicates that a tool ran out of memory during the analysis.

	# of Reported Data Races		
Benchmark	**archer**	**archer-low**	**sword**
miniFE	0	0	0
HPCCG	1	1	1
LULESH	0	0	0
AMG2013_10	4	4	14
AMG2013_20	4	4	14
AMG2013_30	4	4	14
AMG2013_40	OOM	OOM	14

accesses, while SWORD detects them since it logs every memory access.

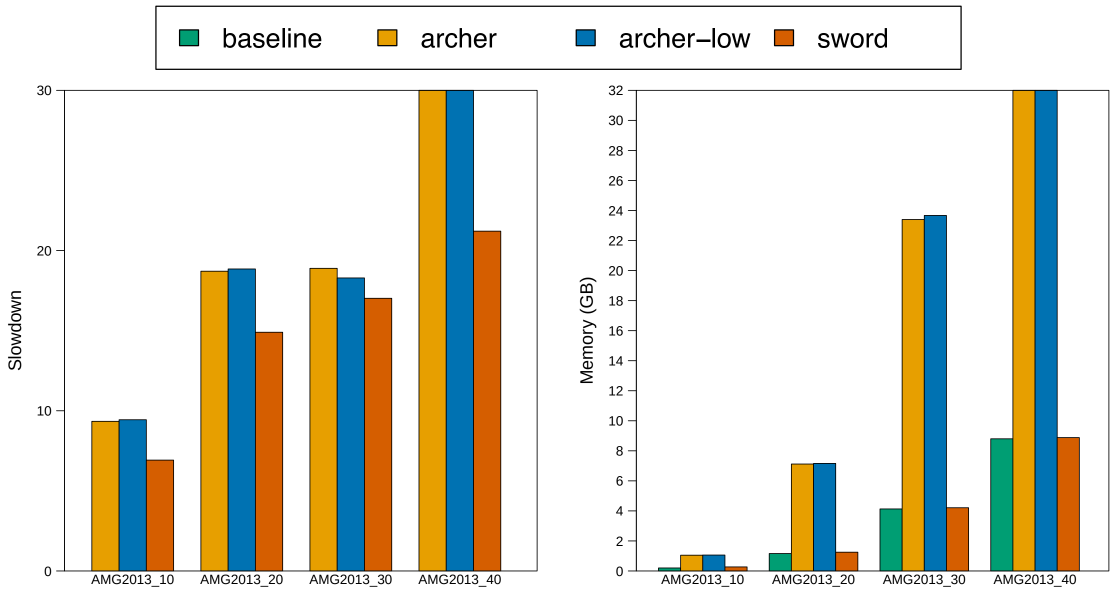
Figure 4.7 shows the slowdown and memory overhead of the tools on the HPC benchmarks. ARCHER in both configurations exhibits a larger slowdown than SWORD as we are increasing the number of threads. The "archer-low" configuration flushes the shadow memory in-between independent parallel regions, and the plots show that this slightly reduces the memory overhead, but it also increases the runtime overhead because of the additional operations to release memory pages. SWORD, on the other hand, exhibits better scaling, typically resulting in a faster dynamic analysis than ARCHER, with the exception of LULESH (see Figure 4.7c). LULESH executes a large number of parallel regions and barriers that significantly increase the number of I/O operations during the log collection phase of SWORD. The plots show that the memory overhead of ARCHER depends on the baseline memory consumption and is around 5–7× of the baseline. On the other hand, SWORD's memory overhead is bounded since it depends only on the number of threads (it is around 3.3 MB per thread) and not the baseline. Figure 4.8 further analyzes this behavior by varying the problem input size of AMG2013. This clearly illustrates a major advantage of SWORD: as the baseline memory consumption increases ARCHER runs out of memory,



(a) miniFE slowdown (b) HPCCG slowdown (c) LULESH slowdown (d) AMG2013_30 slowdown

(e) miniFE memory overhead (f) HPCCG memory overhead (g) LULESH memory overhead (h) AMG2013_30 memory overhead

Figure 4.7: Relative slowdown and memory overhead compared to the baseline for HPC benchmarks



(a) Runtime overhead (b) Memory overhead

Figure 4.8: Runtime and memory overhead on AMG2013 with varying problem size executed with 24 threads.

while SWORD's bounded memory overhead allows it to finish its analysis successfully.

As Figure 4.7 and Figure 4.8 indicate, SWORD's dynamic analysis (log collection) is typically faster than ARCHER at larger scales. However, we need to take the offline analysis execution time into account to represent the total runtime overhead of SWORD. Table 4.5 shows the overheads of the tools including the offline analysis of SWORD. The overall analysis runtime of SWORD for HPCCG, including the offline data race detection process, is less than 2 minutes if executed sequentially and can be reduced to several seconds if executed in parallel; the latter is not significantly different from ARCHER. On the other

Table 4.5: Overheads on the HPC benchmarks executed with 24 threads, including the execution time of the parallel offline analysis. See Table 4.3 for explanation of columns. OOM indicates that the tool ran out of memory during the analysis.

Benchmark	baseline(s)	archer(s)	archer-low(s)	sword				
				DA(s)	OA(s)	MT(s)	#PR	LS(GB)
miniFE	4.7	101.4	101.6	13.3	8.1	4.3	28	1.1
HPCCG	0.4	10.5	46.3	14.4	84.9	2.3	898	2.8
LULESH	3.9	116.1	115.6	131.7	>24h	40.0	300,000	9.8
AMG2013_10	2.2	19.8	20.1	14.9	811.0	5.4	1,272	2.4
AMG2013_20	7.7	149.1	147.2	115.9	2,116.0	41.0	1,527	20.0
AMG2013_30	23.8	471.4	448.2	418.7	3,153.0	133.2	1,575	57.0
AMG2013_40	57.2	OOM	OOM	1,251.4	3,871.0	180.2	2,036	162.0

hand, SWORD is about 4 times faster than ARCHER on miniFE. On LULESH, the dynamic analysis component is comparable for both tools, but the SWORD's offline analysis takes more than 24 hours. The reason is that LULESH generates almost 300,000 independent parallel regions to be analyzed by the offline analysis, which can take a long time, even with parallelization. For our experiments we used 24 cores, each core generating the interval-tree of a different thread. While the tree generation cannot be efficiently parallelized since it would require the use of locks, we could significantly reduce this large offline analysis time by using many more cores for the comparison of the interval trees of different threads. The most interesting case is AMG, where ARCHER runs out of memory at large problem sizes and does not complete its analysis, while SWORD is able to collect all the data at runtime and perform the offline data race detection process. Even though SWORD's offline analysis takes about an hour when executed sequentially, it does not take more than a few minutes when executed in parallel, and the data race detection is more complete than ARCHER.

4.5 Related Work

Data race detection is a widely studied problem in concurrent program design. A good survey of general approaches for data race detection can be found in [7]. A number of different approaches have been taken, including static-analysis [18, 76, 16, 8, 56], dynamic-analysis [10, 66], and hybrid-analysis [14]. These techniques are not directly applicable to OpenMP programs, as they fail to consider the internal actions of OpenMP programs and their runtimes. A complete survey of data race detection methods is beyond the scope of this work; in this section we focus on works that either address OpenMP race checking, or are more closely related.

ARCHER [23, 22], to the best of our knowledge, is the only OpenMP data race detector with enough low runtime overhead that can analyze real-world scientific applications. However, the main weakness of ARCHER is its memory consumption which can reach 6x the amount of memory needed by the application when not being analyzed by the tool. Admittedly, ARCHER provides an option to release some of the analysis memory in between independent parallel regions, reducing the memory overhead over 30%. However, as we show in Section 4.4, ARCHER's memory reduction is not enough to target large

OpenMP applications that allocate up to 90% of the available memory in each compute node.

There have been many efforts that make race-checking efficient by exploiting the structured parallelism found in languages such as Cilk [77], X10 [78], or Habanero Java [27]. These techniques are not directly applicable to OpenMP.

Similarly to SWORD, Wilcox et al. [79] propose an approach to reduce memory overhead by employing array summarization where array accesses can be summarized into the same shadow-cell. This approach reduces the memory overhead by about 30% for array-intensive applications, however it does not overcome the happens-before and shadow-memory limitations explained in Section 4.2.

4.6 Conclusions

Given the growing importance of OpenMP for harnessing on-node parallelism, data races in production-scale OpenMP programs present a looming threat to reliable parallel software design. Today's happens-before relation based race checkers for OpenMP (notably ARCHER, the best in its class) are highly memory inefficient, needing at least five times (and in practice, six times) more memory than the application itself. Despite using this amount of memory, they also miss a significant number of data races due to either schedule-based race masking or shadow-cell eviction.

In contrast, in our new work embodied in the tool SWORD, the online can be carried out using a memory buffer of under 3 megabytes in size. Traces collected in this buffer are compressed, and written out, where an offline analysis based on stepping an operational semantic model takes over. This algorithm is also memory efficient, being based on novel streaming algorithms and state-of-the-art interval tree data structures to merge traces and check for races. Overall, SWORD is at least 1,000 times more memory-efficient than ARCHER, thus virtually guaranteeing the absence of out-of-memory errors. For instance, we could not finish checking the AMG2013 benchmark at large scale using ARCHER, while with SWORD it was easily accomplished.

We present extensive experimental results that demonstrate these features of SWORD as well as its overall superior performance. These experiments were performed on a recently published OpenMP benchmark suite [64] as well as all previous data race checking benchmarks on which ARCHER was run. These experiments demonstrate that SWORD quite favorably matches ARCHER even on examples where the memory pressure is not an issue. SWORD is also sound and complete with respect to data race checking in the absence of data-dependent control flow variations.

While SWORD's dynamic analysis is overall faster than ARCHER, its offline data race analysis can sometimes take a long time, especially at very large scales. This slow-down can be mitigated through the development of novel parallel algorithms (future work). We also plan to extend SWORD's approach to target regions that are offloaded on accelerators, as well as accommodate tasking.

In conclusion, SWORD is currently the tool of choice for checking large-scale OpenMP programs: Through systematic control-flow path coverage, users can detect races more effectively than with available tools, and do not worry about out-of-memory errors even when checking against their users' production inputs.

4.7 Summary

In this chapter we have presented the following contributions:

- A formal semantics based data race checker for OpenMP applications.
- A low overhead technique to log all loads and stores of a program and enable data race analysis.
- An offline data race detection algorithm that guarantee soundness and completeness for a given input.
- An new OpenMP race checker tool that enables data race analysis for large HPC applications where current existing techniques and tools fails.
- An evaluation on a suite of 20 OpenMP microbenchmarks and four real-world HPC applications to indicate the low overhead and effectiveness of the technique.

CHAPTER 5

CONCLUSIONS

This chapter concludes the dissertation. In this work, we presented three contributions to the area of data race detection and OpenMP program analysis. All three contributions extended existing techniques or presented novel approaches to improve accuracy and precision of data race detection while maintaining a low runtime and memory overhead.

Chapter 2 addressed the fact that the state-of-the-art for OpenMP data race detection lacks usable and open-source data race checkers. Existing OpenMP race checkers miss races, report false alarms, and often their high runtime and memory overhead makes them impractical for analyzing large OpenMP applications. We looked into existing static and dynamic analysis techniques to combine the best of these two approaches to obtain a lightweight data race checker called ARCHER. ARCHER builds on the well-known compiler infrastructure Clang/LLVM, it is portable across different operating systems and architecture and is open-source. Its approach is to apply static analysis methods to identify race free regions of code and apply a dynamic analysis technique only to the potentially racy code. The runtime analysis depends on the Clang/LLVM ThreadSanitizer runtime, which we made able to analyze OpenMP application due to an annotation of the OpenMP runtime. Results show that ARCHER is more precise and accurate than existing OpenMP data race detectors, and guarantee low runtime overhead, a critical property to analyze HPC applications. Early usage has shown successes in identifying data races in real-world applications that are paramount at the Lawrence Livermore National Laboratory and other national laboratories across the country.

In Chapter 3 we defined an operational semantics that describes the concurrency structure of OpenMP and enable a more precise and accurate data race analysis. The operational semantics provides a set of rules that capture every event of an OpenMP program and identify races. We show how the semantics improves the data race detection in OpenMP programs by overcoming the limitations of existing techniques such as happens-before. The set of rule is extensible to allow the analysis of future OpenMP constructs, and expand the data race analysis on currently unsupported OpenMP features such as tasking and target devices.

With Chapter 4 we introduced a novel OpenMP data race detection techniques based on the operational semantics illustrated in Chapter 3. The new technique consists of logging at runtime all loads, stores and OpenMP events into files. This information is then analyzed by an offline algorithm which identifies potential data races. We implement this new technique in a tool called SWORD, which has shown in the experimental results to be more precise and accurate than existing data race detection approaches based on happens-before or lockset. The results show that SWORD has almost non existent memory overhead, a property that allows the tool to check a class of OpenMP HPC applications that require a large amount of memory and that with other tools, including ARCHER, would not be possible to analyze because of their high memory overhead.

In these three chapters we demonstrated our thesis that combining the best of existing techniques, and exploiting the concurrency structure of a programming model such as OpenMP, we can make data race checking of HPC applications practical.

REFERENCES

[1] "CORAL/Sierra," https://asc.llnl.gov/coral-info.

[2] "SUMMIT: Scale new heights. discover new solutions." https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf.

[3] "Trinity," http://www.lanl.gov/projects/trinity/.

[4] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[5] "Computer codes," https://wci.llnl.gov/simulation/computer-codes.

[6] M. Süss and C. Leopold, "Common mistakes in OpenMP and how to avoid them: A collection of best practices," in *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming*, 2008, pp. 312–323.

[7] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, no. 1, pp. 74–88, Mar. 1992.

[8] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," in *ESEC/FSE*, 2007, pp. 205–214.

[9] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *PLDI*, 2009, pp. 121–133.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," *SIGOPS Oper. Syst. Rev.*, pp. 27–37, Oct. 1997.

[11] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *The Journal of Supercomputing*, Jul. 1978.

[12] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09, 2009, pp. 62–71.

[13] J. Erickson, S. Freund, and M. Musuvathi, "Dynamic analyses for data-race detection," in *Runtime Verification*, Sep. 2012, pp. 1–1.

[14] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," *SIGPLAN Not.*, pp. 167–178, Jun. 2003.

[15] R. J. Dias, V. Pessanha, and J. M. Lourenço, *Precise Detection of Atomicity Violations*. Berlin, Heidelberg: Springer, 2013, pp. 8–23.

[16] M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," *SIGPLAN Not.*, vol. 41, no. 6, pp. 308–319, Jun. 2006.

[17] C. Flanagan and S. N. Freund, "Type-based race detection for Java," *SIGPLAN Not.*, pp. 219–232, May 2000.

[18] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, pp. 237–252, Oct. 2003.

[19] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, pp. 385–394, Jul. 1976.

[20] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic analysis of concurrency errors in OpenMP programs," in *International Conference on Parallel Processing*, 2013, pp. 510–516.

[21] G. Li and G. Gopalakrishnan, "Scalable SMT-based verification of GPU kernel functions," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, 2010, pp. 187–196.

[22] J. Protze, S. Atzeni, D. H. Ahn, M. Schulz, G. Gopalakrishnan, M. S. Müller, I. Laguna, Z. Rakamarić, and G. L. Lee, "Towards providing low-overhead data race detection for large OpenMP applications," in *2014 LLVM Compiler Infrastructure in HPC*, Nov. 2014, pp. 40–47.

[23] S. Atzeni, G. Gopalakrishnan, Z. Rakamarić, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: Effectively spotting data races in large OpenMP applications," in *IPDPS*, 2016, pp. 53–62.

[24] Center for Applied Scientific Computing (CASC) at LLNL, "Hypre," http://acts.nersc.gov/hypre/.

[25] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly detection of data races in OpenMP programs," in *PADTAD*, 2012, pp. 1–10.

[26] S. Atzeni and J. Protze, "ARCHER, a low overhead data race detector for openmp programs," https://github.com/PRUNER/archer, 2016.

[27] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," in *RV*, 2010, pp. 368–383.

[28] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*. Berlin, Heidelberg: Springer, Sep. 2013, pp. 171–185.

[29] S. H. Langer, I. Karlin, and M. M. Marinak, "Performance characteristics of HYDRA – a multi-physics simulation code from LLNL," in *High Performance Computing for Computational Science – VECPAR 2014*, Jun. 2014, pp. 173–181.

[30] Lawrence Livermore National Laboratory, "National Ignition Facility and Photon Science," https://lasers.llnl.gov.

[31] ——, "Advanced simulation and computing sequoia," https://asc.llnl.gov/computing_resources/sequoia.

[32] H.-J. Boehm, “How to miscompile programs with "benign" data races,” in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar’11, 2011, pp. 3–3.

[33] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[34] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[35] T. Grosser, A. Groesslinger, and C. Lengauer, “Polly — performing polyhedral optimizations on a low-level intermediate representation,” *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, Dec. 2012.

[36] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004, pp. 75–86.

[37] C. Lattner, “LLVM and Clang – advancing compilers and tools,” in *Proceeding of the Free and Open Source Software Developers’ European Meeting*, ser. FOSDEM ’11, Feb. 2011.

[38] K. Serebryany and D. Vyukov, “Sanitizer special case list,” http://clang.llvm.org/docs/SanitizerSpecialCaseList.html.

[39] A. J. Dorta, C. Rodriguez, and F. D. Sande, “The OpenMP source code repository,” in *EMPDP*, 2005, pp. 244–250.

[40] “CORAL Benchmark Codes,” https://asc.llnl.gov/CORAL-benchmarks/.

[41] “Intel OpenMP Runtime Library,” https://www.openmprtl.org.

[42] B. Chapman, G. Jost, and R. V. D. Pas, *Using OpenMP: portable shared memory parallel programming*, ser. Scientific and engineering computation. Cambridge, Massachusetts: The MIT Press, 2007.

[43] J. D. Kelleher, B. M. Namee, and A. D’Arcy, *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. Cambridge, Massachusetts: The MIT Press, 2015.

[44] Center for Applied Scientific Computing (CASC) at LLNL, “AMG2013,” https://codesign.llnl.gov/amg2013.php, 2013.

[45] V. E. Henson and U. M. Yang, “BoomerAMG: A parallel algebraic multigrid solver and preconditioner,” *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, Apr. 2002.

[46] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, “Effective data-race detection for the kernel,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI ’10, 2010, pp. 151–162.

[47] P. Pratikakis, J. S. Foster, and M. Hicks, “LOCKSMITH: Practical static race detection for C,” *ACM Trans. Program. Lang. Syst.*, no. 1, pp. 1–55, Jan. 2011.

[48] P. Godefroid and N. Nagappan, "Concurrency at Microsoft: An exploratory survey," in *Workshop on Exploiting Concurrency Efficiently and Correctly (EC2)*, May 2008.

[49] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 337–348.

[50] M. Das, G. Southern, and J. Renau, "Section based program analysis to reduce overhead of detecting unsynchronized thread communication," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '15, 2015, pp. 283–284.

[51] A. Betts, N. Chong, A. F. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: a verifier for GPU kernels," in *OOPSLA/SPLASH*, 2012, pp. 113–132.

[52] P. Li, G. Li, and G. Gopalakrishnan, "Practical symbolic race checking of GPU programs," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 179–190.

[53] "Enabling ThreadSanitizer on PPC64(BE/LE) platforms," http://reviews.llvm.org/D12841, Dec. 2015.

[54] "CUDA-MEMCHECK Tool," http://docs.nvidia.com/cuda/cuda-memcheck/index.html.

[55] S. Atzeni and G. Gopalakrishnan, "An operational semantic basis for openmp race analysis," https://arxiv.org/abs/1709.04551, Sep. 2017.

[56] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, *An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection*. Rochester, NY, USA: Springer International Publishing, 2017, pp. 106–120.

[57] Tim Lewis, "OpenMP Specifications," http://www.openmp.org/specifications.

[58] Intel, "Intel Cilk Plus," https://www.cilkplus.org, 1994.

[59] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, pp. 519–538, Oct. 2005.

[60] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," *SIGPLAN Not.*, pp. 531–542, Jun. 2012.

[61] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*, 1988, pp. 215–226.

[62] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *11th Australian Computer Science Conference*, 1988, pp. 55–66.

[63] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing*, 1991, pp. 24–33.

[64] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, "Dataracebench: A benchmark suite for systematic evaluation of data race detection tools," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17, 2017, pp. 11:1–11:14.

[65] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–13.

[66] K. Serebryany and D. Vyukov, "ThreadSanitizer, a data race detector for C/C++ and Go," https://github.com/google/sanitizers.

[67] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "Unraveling data race detection in the intel thread checker," in *STMCS*, 2006.

[68] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," *SIGPLAN Not.*, pp. 141–152, Jun. 2013.

[69] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," *SIGPLAN Not.*, pp. 387–400, Jan. 2012.

[70] M. F. Oberhumer, "LZO," http://www.oberhumer.com/opensource/lzo, 2012.

[71] Google, "Snappy," https://google.github.io/snappy, 2011.

[72] Y. Collet, "LZ4," https://lz4.github.io/lz4, 2011.

[73] J. Gama, *Knowledge Discovery from Data Streams*, 1st ed. London, UK: Chapman & Hall/CRC, 2010.

[74] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta Inf.*, pp. 290–306, Dec. 1972.

[75] "Mantevo," https://mantevo.org, 2013.

[76] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *Computer Aided Verification*, 2007, pp. 226–239.

[77] G. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data rase in cilk programs that use locks," in *SPAA*, 1998, pp. 298–309.

[78] T. Yuki, P. Feautrier, S. V. Rajopadhye, and V. Saraswat, "Checking race freedom of clocked X10 programs," *CoRR*, vol. abs/1311.4305, 2013.

[79] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, "Array shadow state compression for precise dynamic race detection," in *ASE*, Nov. 2015, pp. 155–165.