ARCHER: Effectively Spotting Data Races in Large OpenMP Applications

Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamarić
University of Utah
Salt Lake City, UT, United States
{simone, ganesh, zvonimir}@cs.utah.edu

Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee
Lawrence Livermore National Laboratory
Livermore, CA, United States
{ahn1, lagunaperalt1, schulzm, lee218}@llnl.gov

Joachim Protze, Matthias S. Müller
RWTH Aachen University
Aachen, Germany
{protze, mueller}@itc.rwth-aachen.de

***Abstract*—OpenMP plays a growing role as a portable programming model to harness on-node parallelism; yet, existing data race checkers for OpenMP have high overheads and generate many false positives. In this paper, we propose the first OpenMP data race checker, ARCHER, that achieves high accuracy, low overheads on large applications, and portability. ARCHER incorporates scalable happens-before tracking, exploits structured parallelism via combined static and dynamic analysis, and modularly interfaces with OpenMP runtimes. ARCHER significantly outperforms TSan and Intel®Inspector XE, while providing the same or better precision. It has helped detect critical data races in the Hypre library that is central to many projects at Lawrence Livermore National Laboratory and elsewhere.**

***Keywords*-data race detection; OpenMP; high performance computing; static analysis; dynamic analysis;**

I. 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]–[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. We are working with computational scientists at Lawrence Livermore National Laboratory (LLNL), one of the world's largest computing facilities, where many of our mission-critical multiphysics applications [4] are being ported 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 pre-existing 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 [5], 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.

In this paper we describe 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 pre-existing tool—namely ThreadSanitizer (TSan) [6]. 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 [7]. Previous OpenMP data race checking tools were never released for public evaluation, were based on binary instrumentation through PIN [8], or employed symbolic methods [9]. These approaches are neither scalable nor widely portable. ARCHER has been publicly released under the BSD License [10]. (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 [11]), 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 [7]. ARCHER's approach is architected based on the OMPT standard [12] 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 [13] is a large multiphysics application developed at LLNL, which is used for simulations at the National Ignition Facility (NIF) [14] 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 [15], 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 [5] (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 [16]. Removing these benign races fixed the bug. This episode—detailed in Section III-C—clearly shows that effective data race checkers specifically tailored to high-end computing environments are invaluable during critical projects.

II. APPROACH

Figure 1 illustrates how ARCHER implements our tenets by combining well-layered modular static and dynamic analysis stages. In more detail, our static analysis passes [17]–[19] 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 [20], [21] 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 III-B).

As described earlier, we implemented ARCHER using the LLVM/Clang tool infrastructure [22], [23] and the TSan dynamic race checker [6]. On the static analysis side, ARCHER uses Polly [19] 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 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.[1]

A. 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 [24] (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

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

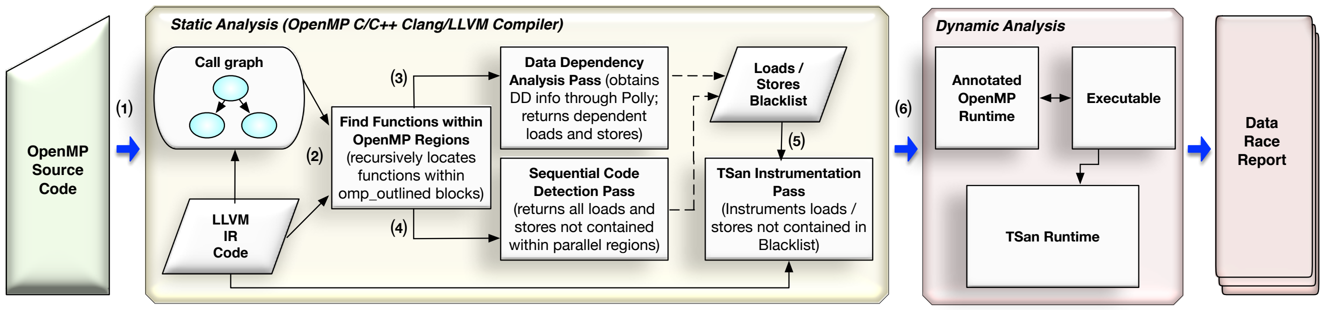

OpenMP Source Code
(1)
Static Analysis (OpenMP C/C++ Clang/LLVM Compiler)
Call graph
LLVM IR Code
(2)
Find Functions within OpenMP Regions (recursively locates functions within omp_outlined blocks)
(3)
(4)
Data Dependency Analysis Pass (obtains DD info through Polly; returns dependent loads and stores)
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 1: ARCHER tool flow.

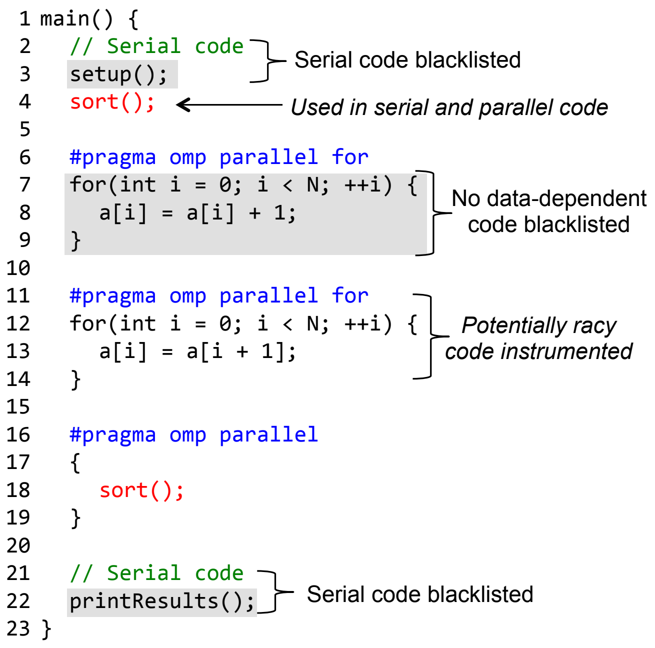
lines. This allows the modified TSan used by ARCHER to exploit our sequential and parallel blacklists, thus guaranteeing a high degree of analysis precision and scalability.

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 1). For the former, an existing tool in the LLVM/Clang suite called Polly [19] is used. In the example given in Figure 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.

ARCHER also identifies sequential code sections (step (4)). In Figure 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.

B. 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 [7].



```
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: Targeted instrumentation on a sample OpenMP program.

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 1).

III. Evaluation

We evaluate Archer in three stages through: (1) a collection of smaller benchmarks called the OmpSCR benchmark suite [25] (an OpenMP source code collection); (2) AMG2013, a non-trivial application from the HPC CORAL benchmark suite [26]; 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.[2] 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 [7], [27]. 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:

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.

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

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

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

Table I: Execution slowdown factor for various tool configurations.

A. OmpSCR Benchmark Suite

We chose the OmpSCR benchmark suite (see Table I) primarily because it harbors a few known races, as reported in prior work [8]. 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_loopA.badSolution`, `c_loopB.badSolution1`, `c_loopB.badSolution2`, `c_testPath`, `c_md`, 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

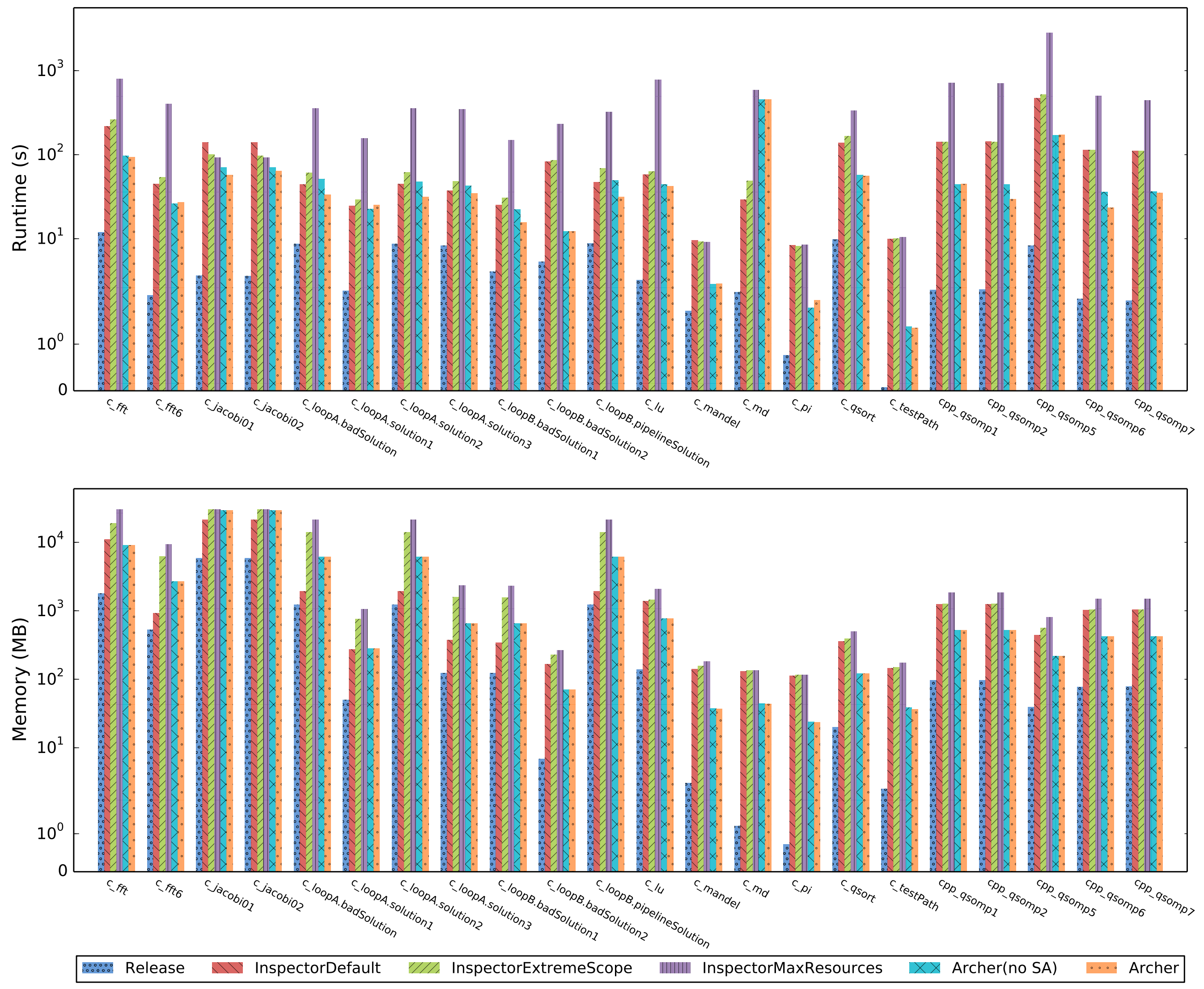


Figure 3: Runtime and memory overhead of the tools on the OmpSCR benchmark suite executed with 16 threads.

contrast, 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[3] 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 [28]. 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

[3]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.

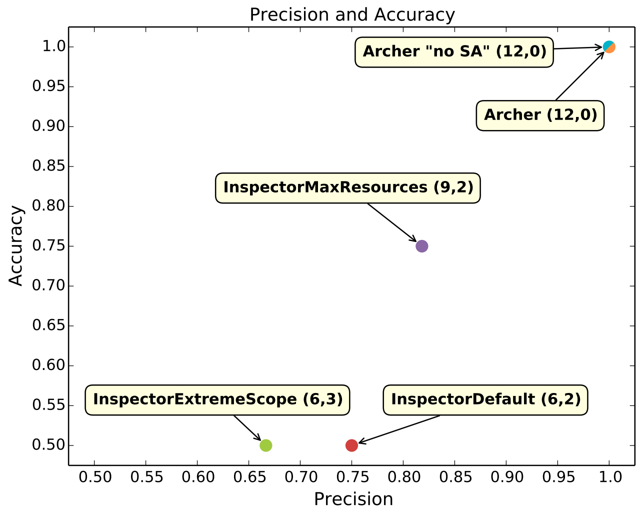


Figure 4: Precision and accuracy; in parentheses we report the number of reported races and false positives.

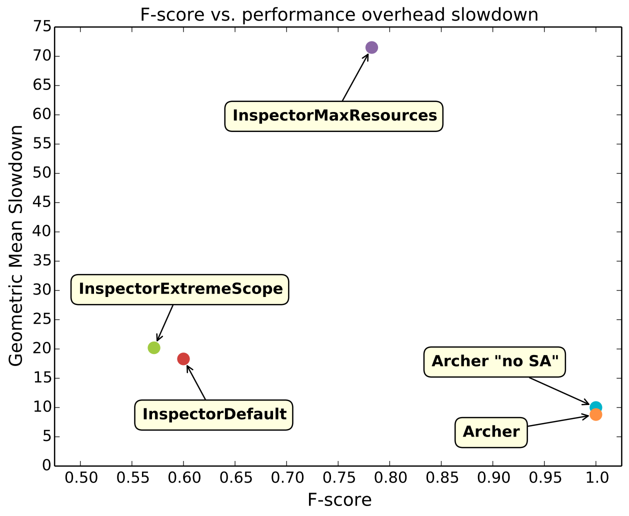


Figure 5: Overall merit expressed as analysis quality vs. performance.

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.[4] Figure 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 III-B, we show that on real-world HPC application static analysis reduces much more the runtime overhead, thus underscoring its importance in practice.

We assess the merits of the tools by plotting their analysis quality against performance. Table I 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 4 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 including Intel®Inspector XE.

In Figure 5, we use an F-score (F1 score) [29] 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 5 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

[4]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.

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.

B. 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 [30] is a parallel algebraic multigrid solver for linear systems and is based on Hypre [5], [31], 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 6 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 6a). However, it is clear that ARCHER has significant performance advantages relative to other tools. In fact, Figure 6b 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 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.

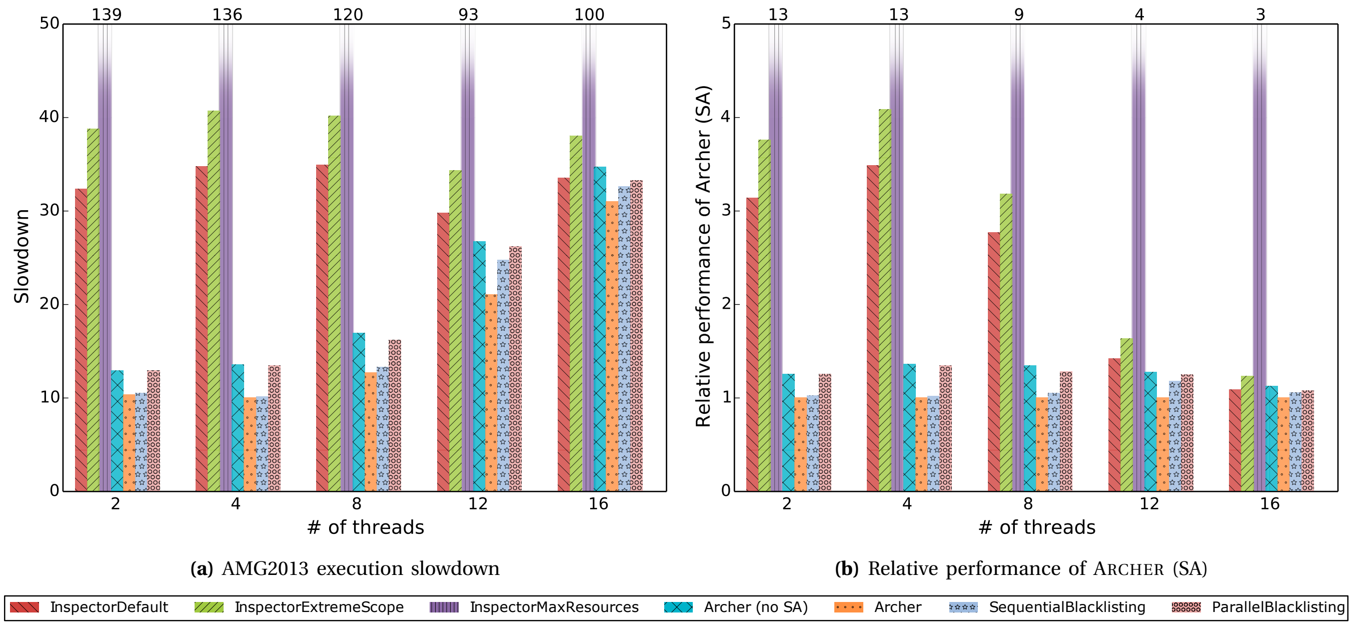
C. ARCHER *Resolves Real-World Races*

We now present how ARCHER aided LLNL scientists in resolving the intermittent crashes in HYDRA mentioned in Section I. 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[5] 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, 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; the 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`.

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



(a) AMG2013 execution slowdown

(b) Relative performance of ARCHER (SA)

Figure 6: AMG2013 execution slowdown factor introduced by the tools (a) and the relative performance factor of ARCHER (SA) against the other tools (b).

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 [16].

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.

IV. 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 [32]; 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. [33], 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 [34], or a misunderstanding of how generated code behaves vis-a-vis the higher level program view (including possible miscompilation [16]). Static race detection methods provide high checking efficiency, but are known to generate false positives (e.g., [35], [36]); each false positive can be a month of wasted reconfirmation time [37]. Polynomial-time race checking can often be achieved under structured concurrency [11]. Predictive methods attempt to find many more "implied" races based on an initial execution through the program (e.g., [38]).

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 [39]. 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 [40]–[42]. This approach is also, in principle, applicable to OpenMP data race checking [9].

V. 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 [43].

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.

A. 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 [44] 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 [42]. 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.

B. Compiler Optimization Flags

Recent work [16] 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.

VI. Conclusions and Future Work

In this paper, we have 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 sub-regions 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 [10], and are looking forward to input from the community.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-PROC-679754). Support is gratefully acknowledged from the National Science Foundation under grants ACI-1535032 and CCF-7298529.

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] "Compute codes," https://wci.llnl.gov/simulation/computer-codes.

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

[6] "ThreadSanitizer, a data race detector for C/C++ and Go," https://code.google.com/p/thread-sanitizer/.

[7] 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 *LLVM Compiler Infrastructure in HPC*, 2014, pp. 40–47.

[8] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly Detection of Data Races in OpenMP Programs," in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2012, pp. 1–10.

[9] H. Ma, S. Diersen, L. Wang, C. Liao, D. J. Quinlan, and Z. Yang, "Symbolic analysis of concurrency errors in OpenMP programs," in *ICPP*, 2013, pp. 510–516.

[10] "ARCHER," https://github.com/PRUNER/archer.

[11] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," *FMSD*, vol. 41, no. 3, pp. 321–347, 2012.

[12] 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*, 2013, pp. 171–185.

[13] S. H. Langer, I. Karlin, and M. Marinack, "Performance characteristics of HYDRA — a multi-physics simulation code from LLNL," Lawrence Livermore National Laboratory, Tech. Rep., 2014.

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

[15] Lawrence Livermore National Laboratory, "Advanced Simulation and Computing Sequoia," https://asc.llnl.gov/computing_resources/sequoia.

[16] H.-J. Boehm, "How to miscompile programs with "benign" data races," in *USENIX Conference on Hot Topic in Parallelism*, 2011, pp. 3–3.

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

[18] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., 1997.

[19] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on low-level intermediate representation," *Par. Proc. Letters*, vol. 22, no. 04, 2012.

[20] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71.

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

[22] C. Lattner, "LLVM and Clang: advancing compiler technology," in *FOSDEM*, 2011.

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

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

[25] A. J. Dorta, C. Rodríguez, F. de Sande, and A. González-Escribano, "The OpenMP source code repository," in *PDP*, 2005, pp. 244–250.

[26] "CORAL Benchmark Codes," https://asc.llnl.gov/CORAL-benchmarks/.

[27] "Intel OpenMP Runtime Library," https://www.openmprtl.org.

[28] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007.

[29] J. D. Kelleher, B. Mac Namee, and A. D'Arcy, *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies.* MIT Press, 2015.

[30] Center for Applied Scientific Computing (CASC) at LLNL, "AMG2013," https://codesign.llnl.gov/amg2013.php.

[31] V. E. Henson and U. M. Yang, "BoomerAMG: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, pp. 155–177, 2002.

[32] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *LOPLAS*, pp. 74–88, 1992.

[33] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *OSDI*, 2010.

[34] J. Erickson, S. Freund, and M. Musuvathi, "Dynamic analyses for data-race detection," http://rv2012.ku.edu.tr/invited-tutorials/, 2012.

[35] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," *TOPLAS*, vol. 33, no. 1, pp. 3:1–3:55, 2011.

[36] 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.

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

[38] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *PLDI*, 2014, pp. 337–348.

[39] M. Das, G. Southern, and J. Renau, "Section based program analysis to reduce overhead of detecting unsynchronized thread communication," in *PPoPP*, 2015, pp. 283–284.

[40] G. Li and G. Gopalakrishnan, "Scalable SMT-based verification of GPU kernel functions," in *FSE*, 2010, pp. 187–196.

[41] 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.

[42] P. Li, G. Li, and G. Gopalakrishnan, "Practical symbolic race checking of GPU programs," in *Supercomputing*, 2014, pp. 179–190.

[43] http://reviews.llvm.org/D12841, 2015.

[44] "CUDA-MEMCHECK tool," http://docs.nvidia.com/cuda/pdf/CUDA_Memcheck.pdf.