ARCHER: Effectively Spotting Data Races in Large OpenMP Applications *

Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamarić
University of Utah

Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee
Lawrence Livermore National Laboratory

Joachim Protze, Matthias S. Müller
RWTH Aachen University

Abstract

Despite decades of research on data race detection, the high performance computing community does not have effective tools to check the absence of races in serial codes being ported over to notations such as OpenMP. The problem lies more with the CS community being unaware of this need, and the HPC community having not faced this need owing to its predominant past reliance on message passing for harnessing parallelism. In this paper, we describe the results of a CS/HPC collaboration through which we have adapted an existing thread-level race checker—namely ThreadSanitizer—to become an effective OpenMP data race checker. Our success is attributable to our having chosen a judicious combination of black-listing of code blocks as well as static analysis to reduce the number of accesses being tracked. In this paper, we report our success in creating a new race checker called ARCHER based on this approach. Our experiments confirm not only the practicality of ARCHER in general terms, but also its ability to shed light on some past incidents of non-determinism observed in critical scientific simulation routines. Specifically, with the help of ARCHER, our team has been able to isolate data races that had vexed engineers. The deeper message in this paper is that of the importance of active collaborations between academic researchers and national lab partners—in many cases more important than technical advances that nevertheless do not transfer to usable tools. We describe ARCHER, its design, its successes, and comment on the future path of our research.

1. Introduction

High performance computing (HPC) is undergoing an explosion in adoption of on-node parallelism. Today's largest systems [8] are significantly underpinned by a plethora of parallelism options on the compute node, including, but not limited to, more cores, wider simultaneous multithreading (SMT), single-instruction/multiple-data (SIMD) units, and accelerators like GPUs. Recent announcements on the next-generation computing systems [6, 7] indicate that the degree of on-node parallelism will be even greater in the future. This trend, in combination with less memory per processing element, is bringing many large production applications, which hitherto have relied solely on MPI, to crossroads where they must transition into hybrid parallelism [1] to realize the full potential of the largest systems.

Due in large part to portability and ease of use, OpenMP offers a highly attractive path to this transition. In fact, the MPI+OpenMP model is becoming increasingly popular even at the largest computing centers. For example, at Lawrence Livermore National Laboratory (LLNL), most of our mission-critical multiphysics applications [4] are currently undergoing this transition, involving teams of programmers to combine large production MPI codes with OpenMP parallelism. However, porting large applications (e.g., over million lines of code [19]) to OpenMP is non-trivial and error-prone, and data races introduced into large code base are extremely challenging to debug.

Broadly speaking, a race checker that can effectively spot a data race in a large code base demands several key attributes. Most of all, the tool should incur low runtime overheads, both in terms of performance and memory, and do this without sacrificing analysis accuracy (i.e., number of true races returned) and precision (i.e., number of found races that can actually occur in practice). In addition, the tool should be portable as the lifetime of these large applications spans many generations of computing platforms. Unfortunately, most exiting tools fall short of satisfying all these attributes, rendering them largely ineffective.

Most dynamic analysis-based tools, like Helgrind [17] and Intel®Inspector XE (hereafter referred to as Inspector), can incur over 30- to 100-fold execution slowdown and over 10x memory overhead on large applications. ThreadSanitizer (TSan) [26] began to break this *runtime-overhead wall* with clever compiler-based instrumentation and shadow-memory access schemes. Similarly to other open-source tools like Helgrind, it only works on low-level threading models (e.g., POSIX Threads) and suffers significant precision losses (i.e., many false alarms) on high-level ones like OpenMP. On the other hand, the attributes of most static analysis tools including Intel Security Static Analysis (SSA) are opposite: while they do not incur runtime overhead, they greatly suffer low analysis precision and accuracy.

In this paper, we present ARCHER, a novel OpenMP data-race checker that fills this gap to aid HPC programmers in this MPI+OpenMP transition. It combines static and dynamic analysis techniques within open-source infrastructure frameworks (LLVM and ThreadSanitizer) to satisfy the key attributes demanded by this transition. ARCHER's LLVM-based static techniques, such as data-dependency and serial-code-detection analyses, first identify those code regions where it can make static race freedom guarantees. Then, it instruments only the remaining, potentially unsafe regions to keep the runtime overheads of its dynamic analysis component at bay.

More specifically, we make the following contributions:

- The first portable OpenMP data race detector that combines static and dynamic techniques, meaningfully lowering runtime overheads without accuracy and precision losses;
- A collection of techniques that enrich and seamlessly bridge static and dynamic techniques in analyzing a large code base;
- An annotation technique that directly integrates *happens-before* knowledge into an OpenMP runtime library allowing runtime detectors to avoid false alarms;

* This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-670108).

- Evaluations including a real-world case study to demonstrate the efficacy and performance of ARCHER.

Our performance evaluation shows that ARCHER significantly outperforms ThreadSanitizer and Inspector, while providing the same or better precision. Our case study shows that ARCHER is capable of discovering previously unknown, elusive OpenMP data races in AMG2013, one of the CORAL benchmark codes [2] as well as the latest release of Hypre [14].We discovered that these races could be linked to some of the intermittent simulation failures that the programmers gave up on debugging in the past.

2. Motivating Example

HYDRA [19] is a large multiphysics application developed at LLNL, which is used for simulations at the National Ignition Facility (NIF) 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 POSIX Threads) in addition to MPI. It has over one million lines of code and a development lifetime that exceeds 20 years.

In 2013, developers began to develop code to port HYDRA to Sequoia [21], 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, they got significantly impeded when they could not resolve a non-deterministic crash on an OpenMP version of Hypre [14] (used by one of HYDRA's scientific packages). The developers had a very hard time to debug this error because it occurred only intermittently, only at large scales (at or above 8192 MPI processes), and only under compiler optimization. After the team had spent substantial amounts of time, they 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.

This case clearly shows that we can benefit from effective data race detectors specifically tailored to high-end computing environments and our findings in Section 4.3 suggest that these tools could have aided the team in quickly testing their suspicion. Unfortunately, a significant gap exists in tools that can aid programmers to debug OpenMP data races in high-end computing environments. Existing static analysis techniques for race detection [11, 22] can find possible bugs, but they are imprecise by nature and produce many false alarms. On the other hand, dynamic analysis techniques [12, 24, 25], produce less (or no) false alarms , but they can miss races and incur high overheads.

Despite all these existing techniques and tools, we have found that none of them is well suited for debugging large OpenMP programs out of box. As an example, consider ThreadSanitizer [25], a state-of-the-art dynamic data-race checker. When we apply ThreadSanitizer to debug classical race cases in a medium-size OpenMP code, it crashes due to resource exhaustion and produces large numbers of false alarms because it does not recognize the OpenMP synchronization semantics. ARCHER uses an improved version of ThreadSanitizer that leverages an annotated version of the Intel OpenMP runtime library to recognize library-level synchronization primitives and to avoid false alarms.

3. Approach

Our approach combines several key techniques to embody the design principles described above. First, we seamlessly combine static and dynamic analysis techniques to lower the runtime overheads without sacrificing high analysis accuracy and precision. Second, we build on portable open-source compiler and dynamic analysis infrastructures to gain portability. Third, we

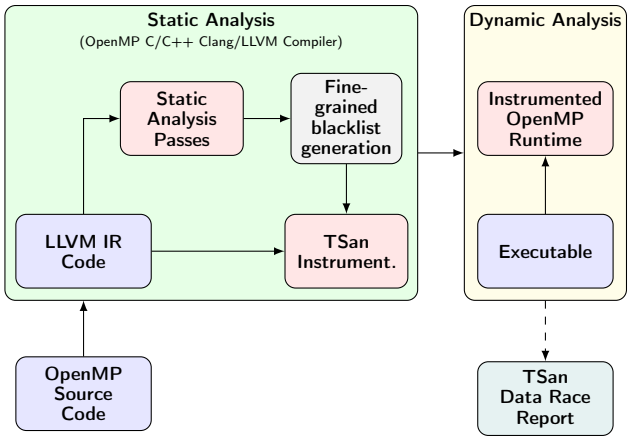


Figure 1: ARCHER tool flow

```
1 main() {
2   // Serial code                    } Serial code blacklisted
3   setup();
4   sort();                           <-- Used in serial and parallel code
5
6   #pragma omp parallel for
7   for(int i = 0; i < N; ++i) {      } No data-dependent
8     a[i] = a[i] + 1;                  code blacklisted
9   }
10
11  #pragma omp parallel for          }
12  for(int i = 0; i < N; ++i) {      } Potentially racy
13    a[i] = a[i + 1];                  code instrumented
14  }                                 }
15
16  #pragma omp parallel
17  {
18    sort();
19  }
20
21  // Serial code                    } Serial code blacklisted
22  printResults();                   }
23 }
```

Figure 2: Targeted instrumentation on a sample OpenMP program

enhance the OpenMP runtime library to expose *happens-before* knowledge explicitly and use that to make our dynamic analysis aware of OpenMP's synchronization semantics.

Figure 1 illustrates our high-level approach embodied in our tool, ARCHER [23], which is split into two phases: static and dynamic. The first phase applies several static analyses [15, 18] on the input program to identify sequential code regions and classify OpenMP code regions into two categories: *race-free* regions and *potentially racy* regions. The second phase leverages the results of the static analysis phase in a dynamic data-race detection algorithm [12, 25] in order to check only the potentially racy OpenMP regions. Leveraging static analysis to dynamically check only the potentially racy regions reduces the runtime and memory overhead of ARCHER, while not degrading its analysis quality.

We implemented ARCHER using the OpenMP [3] branch of LLVM/Clang [20] and ThreadSanitizer dynamic race checker, both open-source tools infrastructures. ARCHER extends some of the static verification passes already present in LLVM and adds some custom passes for further static analyses.

3.1 Static Analysis Phase

ARCHER performs several static analyses on the source files of an OpenMP program to learn conservative information helpful for the subsequent dynamic data-race detection. As shown in Figure 1, an OpenMP program is the expected input to the ARCHER tool flow. The input program is first transformed into LLVM intermediate representation (IR) using the Clang C/C++ front-end. Several LLVM passes analyze IR code to identify code regions that are race-free, are executed sequentially, and to classify OpenMP code regions into the two categories introduced above.

In our work, we leverage an automatic data dependency analysis to identify the OpenMP regions that do not contain data dependencies. Such regions are data race-free, and hence do not have to be further analyzed by ThreadSanitizer, which can in turn focus on the potentially racy regions. ARCHER performs conservative dependency analysis using an existing tool in the LLVM/-Clang suite called Polly [15]. Polly looks for data dependencies inside the OpenMP constructs. Since a dependency can turn into a data race or not depending on the loop input, in order to avoid missing data races, ARCHER applies a conservative strategy classifying a region with a data dependency as potentially racy, so it is analyzed at runtime. Figure 2 shows an example of a data dependency in lines 11–14. In this case, multiple threads may simultaneously access the same array location and cause a data race. Figure 2 shows an example of an OpenMP for-loop with no data dependencies in lines 6–9, where multiple threads always access distinct array locations, this region can be considered race-free.

The information about race-free and potentially racy regions, classified by the dependency analysis, are stored in a list in terms of line numbers in the source code. The lists containing the line numbers of race-free regions are deemed *blacklists* since all the loads/stores listed can be ignored during the dynamic analysis performed by ThreadSanitizer. When the static analysis passes are done, the ThreadSanitizer instrumentation pass instruments the IR code in order to insert the functions needed for catching data races at runtime. Our customized ThreadSanitizer instrumentation pass takes the blacklists as its input to avoid instrumenting the race-free regions as identified by Polly.

ARCHER also applies to the code custom passes to blacklist the sequential code of an OpenMP program to further reduce the amount of load/stores to be checked at runtime. This is quite powerful on a production simulation that can contain a large portion of un-threaded execution (e.g., packages that only use MPI and serial-code segments). ARCHER identifies a sequential load/store as an instruction that is not reachable from an OpenMP construct, neither directly nor by following (nested) function invocations. If a function is invoked both from within and outside an OpenMP region, its instructions are conservatively considered as being executed in parallel. Function `sort()` in Figure 2 is an instance of this case: it is invoked from sequential code on line 4 and from parallel code on line 18. We implemented a custom pass in ARCHER to identify every load/store executed sequentially in a program, and to generate the appropriate blacklist (in terms of source line numbers). The generated blacklist is used as input to the ThreadSanitizer instrumentation pass, which does not instrument those lines of code.

3.2 Dynamic Analysis Phase

ARCHER uses the ThreadSanitizer runtime analysis for the dynamic data race detection of OpenMP programs. The vanilla version of ThreadSanitizer [26] is a data-race detector for `C/C++` and `Go` programs, and it was not explicitly designed to find data races in OpenMP programs. Nevertheless, since OpenMP parallelism is often enabled by a POSIX Thread-based runtime library, the same technique implemented by ThreadSanitizer can typically check OpenMP code as well. Indeed, running vanilla ThreadSanitizer on a simple racy OpenMP program can pinpoint the race, but it also reports many false alarms due to not capturing the OpenMP semantics.

The OpenMP standard specifies several high-level synchronization points. ThreadSanitizer lacks the knowledge about these synchronization points, causing it to generate many false alarms. We use the Annotation API of ThreadSanitizer to highlight these synchronization points within the OpenMP runtime to avoid such false alarms. As part of our previous work [23], we showed this technique eliminates all false alarms in our benchmarks.

Our next step is to combine the ThreadSanitizer dynamic technique with our static analyses in order to reduce the runtime and memory overhead generated during the verification process. The ThreadSanitizer instrumentation involves inserting special function calls for every load and store instruction that gather memory-access information during runtime. ThreadSanitizer provides a feature that allows users to blacklist functions (by their name) that should not be instrumented and that are thus ignored at runtime [27]. We extended this feature to a more fine-grained selection at the level of source lines [23], allowing us to ignore race-free code regions more precisely as determined by static analyses. The output of the ThreadSanitizer instrumentation step is a selectively instrumented binary that interacts with the ThreadSanitizer runtime analyzer when finally executed.

4. Evaluation

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. We evaluate the effectiveness, performance, and scalability of our tool using the OmpSCR benchmark suite [10] as well as AMG2013, a program from the CORAL benchmark suite [2]. We check each benchmark with Inspector, the vanilla version of ThreadSanitizer, and ARCHER with and without static analysis support. The benchmarks are compiled with LLVM for ThreadSanitizer and ARCHER, and with the Intel Compiler for Inspector. For both compilers we set the debug flag (`-g`) to report information about the position of potential races in the code. For ARCHER we set the optimization flag to `-O0` because its static analysis passes could be misled by higher optimization levels. Once the blacklists are created and for all the other cases, we left the default value of the optimization flag for the particular benchmark (for most benchmark `-O2`. Finally, all benchmarks are linked against the Intel®OpenMP* Runtime [5]. The unmodified version of the Intel OpenMP Runtime is used for the regular, Intel Compiler, and vanilla ThreadSanitizer executions, while our instrumented version [23] is used for ARCHER runs. Each benchmark application is executed using 2, 4, 8, 12 and 16 threads.

We configure the tools using various parameters that influence their effectiveness and performance. We run ARCHER in two modes: without static analysis support (no blacklisting of OpenMP-race-free regions) and with our static analyses. Vanilla ThreadSanitizer is executed using default values for its parameters. For our evaluation, we choose a base configuration for Inspector that outputs a complete report including locations of potential data races (`-collect ti3`), in order to be comparable with ARCHER and ThreadSanitizer, which produce such information by default. Starting from this configuration, we run Inspector in three different modes: *default*, we do not change any other parameters; *scope* (`-knob scope=extreme`), sets the memory-access granularity to 1 byte (same as ThreadSanitizer); *use-maximum-resources* (`-knob use-maximum-resources=true`), detect more data races but does not optimize memory consumption and performance.

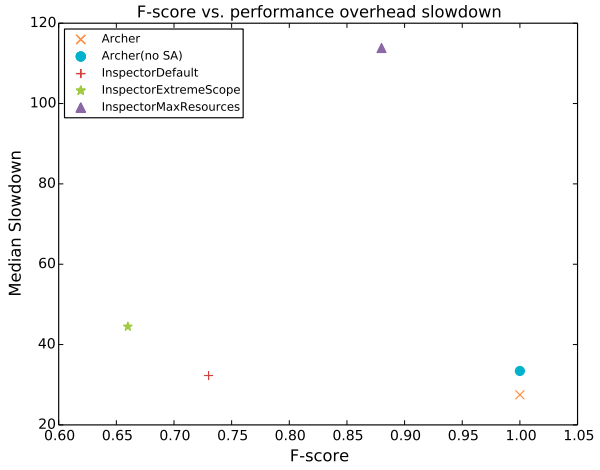


Figure 3: Overall merit on analysis quality vs. performance

4.1 OmpSCR Benchmark Suite

Our experimental results compare the analysis quality and performance of ARCHER with the other tools using the OmpSCR benchmark suite [10]. We chose this suite because it has a known number of races from previous work [16]. For space reasons we omit the runtime overheads for each tool configuration, but we summarize the overall merit of these tools by plotting their *analysis quality vs. performance*.

Our evaluation shows that ARCHER in both configurations (with and without static analysis support) is capable of detecting *all* of the documented races in most of the applications. By contrast, in all three configurations Inspector incurs varying degrees of accuracy and precision loss. In terms of accuracy (the number of correctly detected races divided by the number of true races that should have been detected), Inspector in default configuration misses races in three applications. Its scope-extreme configuration also misses all these races and oddly also misses one additional race. However, Inspector in max-resources configuration detects all of the races. In terms of precision (the number of correctly detected races divided by the number of all the races detected, including false alarms), ARCHER in both configurations incurs no false alarms, while Inspector shows some losses.

We show the merits of these tools by plotting their analysis quality vs. performance. In Figure 3 we use F-score to capture the overall quality of analysis. It is a measure of analysis quality that accounts for both accuracy and precision and is given by:

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

Thus, F-score reaches its best value at 1 and worst at 0. In Figure 3, we plot each tool onto the two-dimensional space defined by F-score and slowdown medians. We use the median as our performance metric because the means are significantly skewed by large data points. This plot shows the general attributes of each tool in terms of accuracy and runtime overheads. Closer is the point to the lower right corner of the plot better are the performance and accuracy/precision of the tool. The plot clearly shows that ARCHER meets our design goal, compared to other state-of-the-art tools: in both configurations it does much better than Inspector in all its configurations.

4.2 AMG2013

To complement our OmpSCR study with a larger code base, we perform our evaluation on AMG2013 (with about 75,000 lines of code), an important program from our CORAL benchmark suite. AMG2013 [13] is a parallel algebraic multigrid solver for linear systems, based on Hypre [14], a large linear solver library developed at LLNL. We ran the three tools on AMG2013, and ARCHER discovered three races unknown to us and never reported before. Vanilla ThreadSanitizer, after reporting about 150 false alarms, crashes and never finishes the verification process. Inspector reports all three data races only when it is configured to *use-maximum resources*. When using the *scope-extreme* configuration, it reports all the three races only when running with 16 threads. Finally, at *default* configuration, Inspector always misses one specific race of the three.

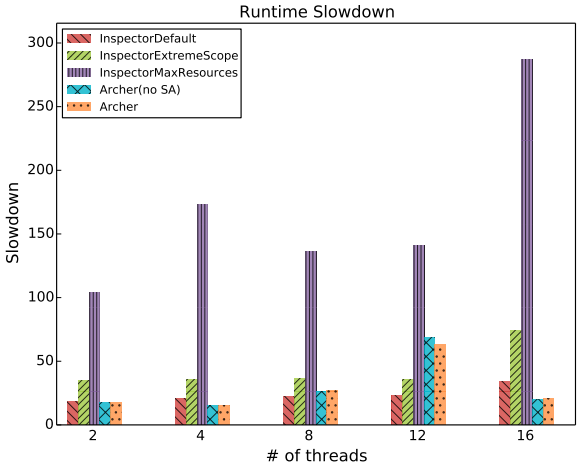
Some of the tools, as dynamic checkers, significantly slow down the program as shown in Figure 4a. However, it is very clear that ARCHER has significant performance advantages relative to other tools. In fact, Figure 4b shows the relative performance of ARCHER (with and without static analysis support) against all of the three configurations of Inspector. Inspector in the default mode outperforms ARCHER at certain thread counts and in *extreme scope* it performs better at 12 threads. As we noted before, however, the performance gain comes at the expense of precision loss. Figure 4b shows that ARCHER is generally 2x-15x faster than Inspector depending on the number of threads. ARCHER compared with itself without any static analysis support improves the performance by about a factor of 1.5.

ARCHER also reduces the memory overhead relative to Inspector in *comparable* configurations, but it is still shown to incur relatively large memory footprints. Our further analysis suggests that this is because ThreadSanitizer's runtime, which ARCHER leverages, allocates shadow memory when the target program accesses that memory first time. For example, an array initialization would access the entire memory allocated to that array and ThreadSanitizer allocates the shadow memory for this array during the initialization. As part of our future work, we plan to advance ways to reduce the memory overheads further, which includes a static mechanism to identify array initializations and to exclude those regions using our blacklisting so as to avoid this *bulk shadow-memory allocation* problem at runtime.

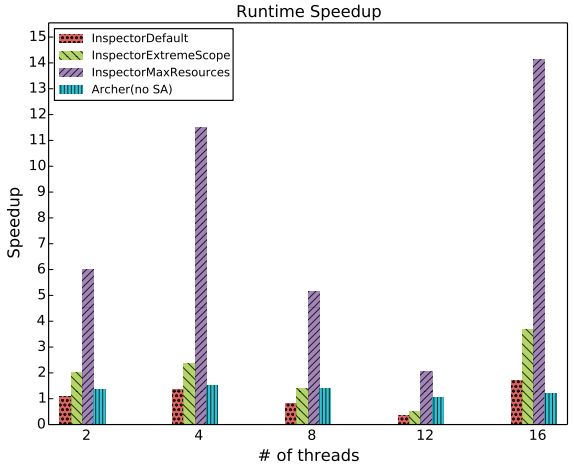
4.3 ARCHER Helps Combat Real-World Races for HYDRA

We present our investigation on applying ARCHER to the intermittent crash issue on HYDRA described in Section 2. We started to make a connection between our findings on AMG and this unresolved real-world issue when we shared these findings with one of the Hypre developers. Of three data races flagged by ARCHER, two cases were found in a fairly complex OpenMP region spanning over 400 source lines with tens of reaching variables. The developer confirmed that both are indeed true data races as a thread will access the first element of a portion of the arrays which belongs to the next thread, and the next thread will subtract a number from this element. However, because the number being subtracted for this particular element is zero, this condition has never been detected through the runtime testing coverage. While programmers often consider this type of races—multiple threads writing the same value to the same memory location—benign, the developer noted that the containing function was one of the routines where they had to disable OpenMP.

Encouraged by our findings, the application team resumed their debugging of this issue. They created a patch to fix these *benign* data races, applied it to the latest Hypre release (2.10.0b), and ran the same simulation with this patched Hypre. This time the simulation failed in a different way: a crash occurred very quickly and much more deterministically. So, we applied ARCHER to the latest Hypre using a representative test program that the developer provided, and ARCHER reported 20 races across



(a) AMG2013 execution slowdown



(b) Relative performance of ARCHER (SA)

Figure 4: AMG2013 execution slowdown caused by the tools and the relative performance of ARCHER (SA)

three routines. The developers will have to analyze our findings more closely, but our initial observation indicates that most of these races again appear to be what most programmers consider benign. If the detected races were indeed the root cause of these crashes, we suspect that the compiler (IBM XL) on this platform, which would assume race-free code for optimization, transformed the code in a way to turn those benign races into malignant ones [9]. Whether they are the root cause or not, it is clear that removing these races leaves such a deep debugging effort with one less unknown, and makes the code more compiler-optimization-safe [9] in the future.

5. 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 brings the best from static and dynamic techniques and seamlessly combines them to deliver on these principles. Our evaluational 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 the scientists' productivity.

However, 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 taping 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 pursue. The adequately defined symmetries will allow ARCHER to target a smaller set of representative threads and memory space for further overhead reduction. Perhaps more importantly and urgently, our team will soon contribute ARCHER back to the open-source community. It is our hope that this tool will help HPC softland this rather disorderly transition to hybrid parallelism.

References

[1] Compilers and more: Mpi+x. `http://www.hpcwire.com/2014/07/16/compilers-mpix/`.

[2] Coral benchmark codes. `https://asc.llnl.gov/CORAL-benchmarks/`.

[3] Openmp/clang. `http://clang-omp.github.io`.

[4] Compute codes. `https://wci.llnl.gov/simulation/computer-codes`.

[5] Intel openmp runtime library. `https://www.openmprtl.org`.

[6] Coral/sierra. `https://asc.llnl.gov/coral-info`.

[7] Summit: Scale new heights. discover new solutions. `https://www.olcf.ornl.gov/wp-content/uploads/2014/11/Summit_FactSheet.pdf`.

[8] Top 500. `http://www.top500.org/`.

[9] H.-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar'11: Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, 2011.

[10] A. J. Dorta, C. Rodríguez, F. de Sande, and A. González-Escribano. The openmp source code repository. In *PDP*, pages 244–250, 2005.

[11] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[12] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[13] C. for Applied Scientific Computing (CASC) at LLNL. Amg2013. `https://codesign.llnl.gov/amg2013.php`, .

[14] C. for Applied Scientific Computing (CASC) at LLNL. Hypre. `http://acts.nersc.gov/hypre/`, .

[15] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – Performing Polyhedral Optimizations on Low-Level Intermediate Representation. *Parallel Processing Letters*, 2012.

[16] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun. On-the-fly detection of data races in openmp programs. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 1–10, 2012.

[17] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. Helgrind+: An efficient dynamic race detector. In *IPDPS*, 2009.

[18] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. 2002.

[19] S. H. Langer, I. Karlin, and M. Marinack. Performance characteristics of hydra - a multi-physics simulation code from llnl. Technical report, 2014.

[20] C. Lattner. Llvm and clang: advancing compiler technology. *Proc. of FOSDEM*, 2011.

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

[22] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.

[23] 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 *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, pages 40–47, 2014.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, pages 391–411, 1997.

[25] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[26] K. Serebryany and D. Vyukov. ThreadSanitizer, a data race detector for C/C++ and Go. `https://code.google.com/p/thread-sanitizer/`, .

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