Archer: A Low Overhead Data Race Detector for OpenMP

Simone Atzeni, Ganesh Gopalakrishnan (Advisor), Zvonimir Rakamarić (Advisor)
School of Computing – University of Utah
Salt Lake City, Utah 84103
{simone, ganesh, zvonimir}@cs.utah.edu

I. Summary

OpenMP is central to portable shared memory parallel programming - yet, porting large HPC applications to OpenMP is error-prone. Data races [1] are particularly egregious. Conventional testing is ineffective for locating data races. Undetected data races are especially problematic for large applications, leading to show-stopper bugs.

Existing approaches to OpenMP data race detection rely on pure static or dynamic analysis techniques. While effective on small- to medium-sized applications, pure approaches are unable to locate races in non-trivial HPC applications - especially those that involve hybrid concurrency. While runtime approaches are known for their accuracy, even state-of-the-art techniques [2], [3] can incur a factor of 100 of slowdown and over 8 times memory overhead on large HPC applications. While static analysis based techniques incur little runtime overhead, they are imprecise and can produce false positives or miss errors.

We propose a novel combined static and dynamic approach suitable for large-scale HPC applications that significantly reduces analysis overhead without sacrificing precision. It first applies a set of static analyses such as loop-carried data dependency and threads-escape analysis to classify OpenMP codes. It then passes potentially unsafe regions into our runtime analyzer. We prototyped a tool, Archer, which extends LLVM and Google's ThreadSanitizer [3] (TSan) to implement this approach.

Figure 1 illustrates our overall approach. The set of source files of an OpenMP application is the input to our system. And the output is a comprehensive data race report. Archer composes static analyses with dynamic techniques to create a seamless analysis workflow.

On the static side, it builds on the Clang/LLVM suite and utilize some of static and dynamic verification passes already present in LLVM including data-dependency analysis [4], loop-carried data-dependency analysis [4], and thread-escape analysis. Specifically, once the OpenMP code is translated in LLVM IR language, Archer applies a collection of static techniques to classify threaded code into three different categories: race-free regions; certainly racy regions; and potentially racy regions (i.e., gray area).

Figure 2a shows a simple OpenMP code where our static data dependency analysis can guarantee data-race freedom: with no data dependency threads *will* each exclusively access distinct array locations. On Figure 2b, data-dependency analysis discovers that this loop has a loop-carried data dependency. This means that threads *may* simultaneously access the same array location and cause a data race. For potentially racy regions like this, our instrumentation framework will add runtime checking code so that they can be further analyzed at runtime for more accurate diagnosis.

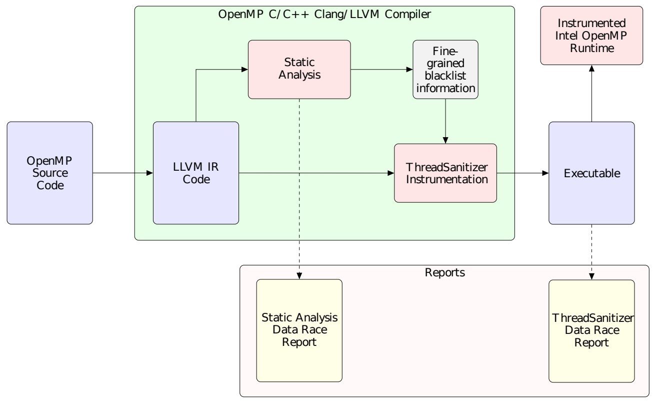
OpenMP C/C++ Clang/LLVM Compiler
Static Analysis
Fine-grained blacklist information
Instrumented Intel OpenMP Runtime
OpenMP Source Code
LLVM IR Code
ThreadSanitizer Instrumentation
Executable
Reports
Static Analysis Data Race Report
ThreadSanitizer Data Race Report

Fig. 1: Overall approach of Archer

Finally, our runtime analysis component is based on TSan. We extended TSan's blacklist feature so that it can exclude code from being examined during runtime at finer granularity (e.g., at the line-number and OpenMP region level) than its original granularity (i.e., function level). Further, because TSan does not support OpenMP, we also instrumented the Intel OpenMP Runtime to allow TSan to recognize its synchronization primitives.

To evaluate the runtime performance and memory benefit of our approach, we use the blacklist feature of TSan and analyze at runtime progressively larger amounts of code in AMG2013, a medium-sized CORAL benchmark code [5] occupying 74,000 lines of code. Table I summarizes our results. These results suggest that the runtime performance overhead of Archer is largely proportional to the number of selected lines of code. However, memory overhead reduction is measured to be less dramatic. We theorize that

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-ABS-658420).

I would like to thank Dong H. Ahn (Advisor at LLNL), Ignacio Laguna (Advisor at LLNL), Greg Lee and, Martin Schulz from Lawrence Livermore National Laboratory for the advice and the support during my internship at LLNL. Furthermore, I want to express my appreciation to Joachim Protze from RWTH Aachen University for his work on the Intel OpenMP Runtime.

TABLE I: Performance Evaluation Results of ThreadSanitizer

% of Instrumented LOC	release	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
time (s)	39.08	387.92	736.75	1020.80	1277.14	1588.87	1786.95	2176.08	2393.82	2627.44	2948.92	3303.80
mem (GB)	3.37	10.71	16.50	19.69	20.99	19.71	20.99	22.52	22.51	22.52	22.52	22.58

```
#pragma omp parallel for
  for(int i = 0; i < N; ++i) {
    a[i] = a[i] + 1;
}
```

(a) data-race free region

```
#pragma omp parallel for
  for(int i = 0; i < N; ++i) {
    a[i] = a[i + 1];
  }
```

(b) Potential data races due to loop-carried dependency

Fig. 2: OpenMP parallel for loop

the reason is twofold. First, the lines of code selection to be instrumented can be considered *random*. Second, TSan shadow memory allocation, during the runtime analysis, happens when the application memory is first touched. This fact together with the random selection of lines of code explains why the memory overhead is not proportional to the amount of instrumented code as happen for the runtime. For example, an array initialization will touch its entire memory originating an allocation of shadow memory by TSan for the entire array memory space. However, in a real usage of Archer, where the blacklisting information are result of static analysis, the instrumented line of code will belong to potentially racy code regions which most likely will not touch the entire allocated memory space generating less overhead. To understand the tool's effects on larger production applications fully, we plan to conduct a comprehensive performance and memory overhead evaluation. We also plan to evaluate Archer on the entire CORAL benchmark suite to gain insight about average code exclusion rate.

To evaluate the effectiveness of our approach, we also conducted a case study where we applied Archer to a development version of Lulesh 2.0, which has a data race error. It is a known error as it had already been fixed in the 2.0 release. Our study shows that excluding lines of race-free code detected by our static analyses still allow TSan to identify the race as accurately.

While our preliminary evaluation suggests promising results, much work still lie ahead of us. In addition to further evaluate overhead and effectiveness (e.g., accuracy and precision) of Archer, we must explore other dimensions beyond the code space where we can apply our targeted approach. Specifically, it is part of our study plan to apply temporal-slice sampling (e.g., periodically analyzing targeted regions at runtime only during particular windows of execution). It is our hope that such techniques can take advantage of periodicity in most of HPC applications and drastically reduce the amounts of analysis that our runtime analyzer must perform while keeping its high accuracy.

References

[1] M. Süßand 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*, IWOMP'05/IWOMP'06, (Berlin, Heidelberg), pp. 312–323, Springer-Verlag, 2008.

[2] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 121–133, ACM, 2009.

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

[4] T. GROSSER, A. GROESSLINGER, and C. LENGAUER, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, 2012.

[5] V. E. Henson and U. M. Yang, "Boomeramg: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2000.