



Understanding the Performance of Dynamic Data Race Detection

Joachim Protze 
IT Center
RWTH Aachen University
Aachen, Germany
protze@itc.rwth-aachen.de

Isabel Thärigen 
IT Center
RWTH Aachen University
Aachen, Germany
isabel.thaerigen@rwth-aachen.de

Jonas Wahle
IT Center
RWTH Aachen University
Aachen, Germany
jonas.wahle@rwth-aachen.de

Abstract—With increasing per-node concurrency, the interest in dynamic data race detection for OpenMP applications increased significantly in recent years. Benchmarks such as DataRaceBench (DRB) help evaluate the classification quality of data race detection tools for simple memory access patterns. Various publications use short-running benchmark kernels from OmpSRC and DRB also for performance benchmarking of data race detection tools. Due to the short execution time, one-time initialization overhead dominates the measurement. Such results are not representative for the overhead with real codes. This paper proposes a new problem class for the SPEC OMP 2012 benchmark designed to analyze the runtime overhead of data race detection tools.

Prior work reported runtime overheads of $80\times$ and higher for the OpenMP data race detection tool Archer (i.e., execution time with the tool is 80 times as long as without a tool). For a specific application, we report $500\times$ runtime overhead in this paper. This overhead stands in contrast to the $2\text{--}20\times$ runtime overhead claimed by the underlying tool ThreadSanitizer. We use our newly proposed input data set to observe and investigate significant runtime overhead of dynamic data race detection for specific applications. With the help of performance analysis tools and hardware performance counters, we can identify massively concurrent read accesses of the same shared variable as the root cause. We identify parallel matrix-vector multiplication as an application pattern responsible for such huge runtime overheads in data race analysis. Finally, we propose a modification of ThreadSanitizer, limiting the runtime overhead for these applications to less than $40\times$.

Index Terms—OpenMP, data race, benchmark, performance

I. INTRODUCTION

Papers presenting tools for dynamic data race detection such as ROMP [1], Archer [2], LLOV [3], or Sword [4] regularly use OmpSCR [5] as a benchmark to evaluate and compare runtime overheads. Unfortunately, in several cases, the input size used is not documented. Some of the papers also use DataRaceBench (DRB) [6] to evaluate the runtime overhead of different detection tools. Calculating and comparing runtime overhead based on the short execution time of OmpSCR and DRB is of limited value. The experiments measure and compare constant, one-time initialization overhead rather than dynamic runtime overhead.

In a more recent Archer paper [7] we started to use SPEC OMP 2012 [8], a well-known and accepted suite of benchmarking applications, with `train` input data to evaluate

the runtime overhead. This predefined input data set does not completely overcome the issue of short execution times. Therefore, we propose a new set of input data for SPEC OMP 2012 in Section II.

The Archer paper also mentions the cost of annotating OpenMP synchronization as a possible source for huge runtime overheads when scaling up the number of threads. Based on our new input data set, we apply performance analysis techniques like tracing, profiling, and measuring hardware-performance counters to identify the actual root cause of the severe runtime overheads. We present the key techniques as well as some results in Section IV.

Based on our findings, we propose a heuristic optimization for the ThreadSanitizer implementation, which is the analysis backend for Archer, in Section V. We also discuss the impact of such optimization on the accuracy of the tool. Finally, we present and discuss performance results for our heuristic optimization in Section VI.

To summarize, the contributions of this paper are:

- the definition of a novel input data set for SPEC OpenMP 2012 suited for performance comparison and analysis of data race detection,
- a performance analysis workflow to identify performance issues in dynamic data race detection of OpenMP applications, and
- a heuristic optimization to address a significant performance issue of ThreadSanitizer when applied to some classes of HPC applications.

II. TOOL OVERHEAD BENCHMARK

In contrast to DataRaceBench as a benchmark for the classification quality of data race detectors for OpenMP, there is no standard benchmark available to evaluate the analysis overhead. Several recent publications about dynamic data race detection use OmpSCR [5] for their performance evaluation. The fundamental issue of this benchmark is that the codes are relatively trivial and try to showcase best practices for OpenMP from 2005 without evolving with the OpenMP standard. The applications also have very short execution times. A better choice might be a benchmark suite like SPEC OMP 2012, which collects a wide variety of actual OpenMP applications. SPEC OMP 2012 so far provides three input

sizes, namely **ref**, **train**, and **test**. With possible runtime overheads of $20 - 100x$, the **ref** size of SPEC OMP does not provide a reasonable base runtime, as each application runs about 20 minutes when executed with 12 threads. In a previous paper [9] we used **train** size for overhead analysis. However, we observed a huge range of three orders of magnitude for the execution times of the different applications, with times ranging from .14 to 181 seconds. Figures 1a/b) highlight the range of three orders of magnitude for **train** and **test** input sizes. In both figures we show the results of 5 repetitions for each configuration. All codes were run with 12, 24, and 48 threads. For linear scaling the execution time should go down by half the distance between the horizontal lines. Several of the applications like 359.**botsspar**, 362.**fma3d**, and 371.**applu331** show bad scalability even for the larger **train** input set.

To study the runtime overhead of data race analysis, we propose a new problem size for the SPEC OMP benchmark targeting about 5 – 10 seconds execution time with 12 threads. Even with $100x$ runtime overhead from a tool, the resulting execution time will be below 20 minutes. On the other hand, an execution time of 5 – 10 seconds also allows to amortize some one-time initialization overhead and also allows to perform performance analysis. The appendix provides all necessary information to derive our new **drd** input size from the existing input sizes. Figure 1c) shows that the execution time for our proposed input data varies by less than a factor of two for 12 and 24 threads. The variation for 48 threads is a bit higher because 362.**fma3d** does not scale to this number of threads. In this figure, we used 36 repetitions for each configuration, showing more details in the variation behavior for each code. Since the y-axis is zoomed by a factor of 4, also the deviation for each configuration appears four times as large as in the two figures below.

III. EXPERIMENT SETUP AND METHODOLOGY

All experiments are executed on exclusively reserved nodes of the Claix 2018 cluster with two Intel Xeon Platinum 8160 processors and 192GB main memory. Hyperthreads are disabled; therefore, a node has 48 cores. Furthermore, sub-NUMA clustering is enabled, leading to four NUMA nodes where the two NUMA nodes within a socket are closer to each other than to the other two NUMA nodes. Unless specified otherwise, threads are pinned using `OMP_PLACES=cores` and `OMP_PROC_BIND=close`. Therefore, experiments with 6, 12, and 24 threads execute on a single socket. Only 48 threads make use of the two sockets. Due to a bug in the OpenMP runtime and a non-obvious numbering scheme of the cores within a socket, the `close` binding does not fill one sub-NUMA node before filling the other one but instead fills both sub-NUMA nodes of one socket alternately.

We use the JUBE benchmarking environment [10] to make the collection of performance data reproducible and also automate the workflow of spawning the measurements for the individual experiments. Furthermore, we use some custom-built versions of LLVM/clang derived from the **main** branch (**30fbb069**) after the recent 13.0 release branch. We

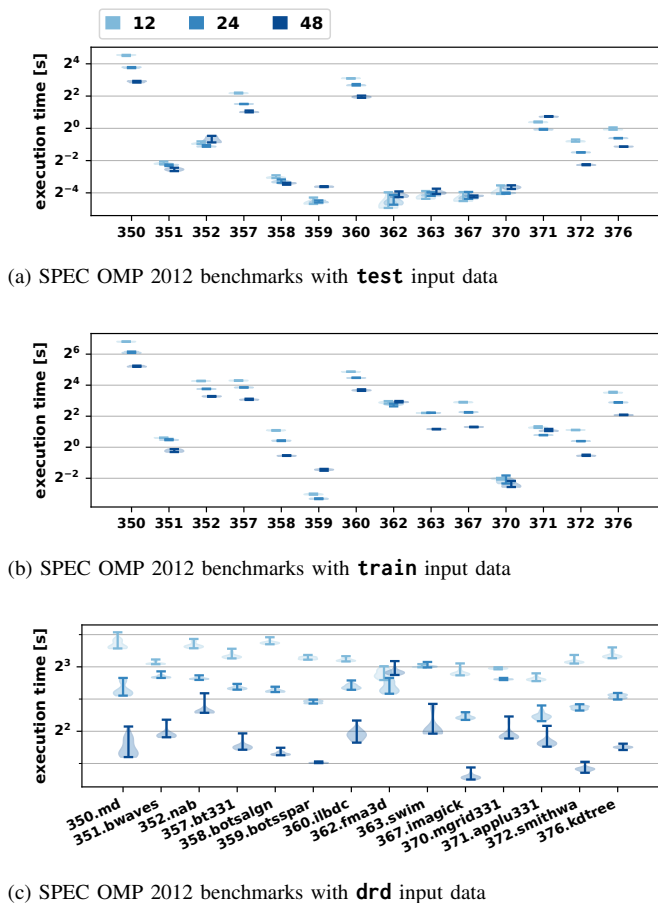


Fig. 1: Execution time of SPEC OMP 2012 benchmarks 12, 24, and 48 threads.

use **gfortran** release version 7.3.0 to compile Fortran codes. To ensure that OpenMP runtime and ThreadSanitizer runtime libraries from LLVM are used, we use clang as linker driver in all cases as suggested in Archer’s documentation¹.

Since LLVM release 10, the OpenMP runtime automatically loads Archer if the application was compiled with ThreadSanitizer support. The difference between Archer (**archer**) and ThreadSanitizer (**tsan**) experiments throughout this paper is that for ThreadSanitizer experiments, we explicitly disable Archer from being loaded or active using the environmental variable `env ARCHER_OPTIONS="enabled=0"`. ThreadSanitizer documentation² suggests to use `env TSAN_OPTIONS="report_bugs=0"` for benchmarking purposes. We use this environmental variable throughout all experiments. This setting allows benchmarking the data race detection overhead rather than the locking and I/O overhead caused by reporting data races. Setting this flag only skips the reporting, while ThreadSanitizer would nevertheless be able to report the number of detected data races or that no race was detected. In a typical debugging workflow,

¹<https://github.com/llvm/llvm-project/tree/main/openmp/tools/archer>

²<https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags>

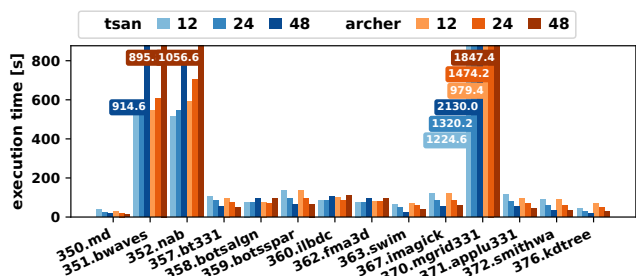


Fig. 2: Execution time of SPEC OMP 2012 benchmarks instrumented with ThreadSanitizer or Archer for 12, 24, and 48 threads.

one would typically abort the execution once several data races were reported to fix the code and then rerun the analysis. The huge runtime overhead caused by suppression of repeated data race reports is not relevant for such workflow.

IV. OVERHEAD ANALYSIS

To the best of our knowledge, no performance analysis study for data race detection was yet presented, going beyond the comparison of total runtime or overhead between different tools. In this study, we want to look into the actual root causes of severe runtime overheads. Therefore, we present our performance analysis approach for data race detection tools, applied to Archer and ThreadSanitizer as two example tools. We chose these two tools because they are very much the same tool allowing a differential performance analysis. As Figure 2 shows, the execution time for the two tools is pretty much the same in many cases. In some cases, execution time with Archer is higher, and in few cases, such as 350.md or 357.bt331 even lower. Since 351.bwaves, 352.nab, and 370.mgrid331 show the highest runtime overheads, we focus our analysis on these three applications.

The compiler pass of ThreadSanitizer instruments all memory accesses in application code to call into ThreadSanitizer’s runtime library and analyze the access for data races. Additionally, ThreadSanitizer intercepts pthread functions and derives the pthread-specific synchronization information from these function calls.

Archer feeds OpenMP synchronization semantics into the vector clock-based happens-before analysis of ThreadSanitizer. In an earlier paper [9], we stated that the additional vector clock exchanges to implement the happens-before analysis might cause the runtime overhead in Archer when scaling to larger thread counts.

A. Identifying performance bottlenecks

To understand the impact of a data race detection tool on the runtime behavior of an application, we need to distinguish the execution of application code from the execution of tool code. Since the data race detection tool instruments and analyses each individual memory access, the execution frequently changes from application to tool code. Due to the granularity of the runtime calls, it is infeasible to track each individual ThreadSanitizer instrumentation of memory accesses. Possible

sources of runtime overhead are 1) additional instructions to implement the tool analysis, 2) additional memory utilization of the tool leading to cache eviction, 3) additional branch instructions resulting in pipeline stalls. Therefore, we see the need to collect hardware performance counters for the individual parts of the execution to understand the impact of instrumentation, logging, and vector clock exchange at the hardware level.

A common performance analysis approach to overcome the granularity issue is to collect performance information using sampling as, for example, preferred by HPCToolkit [11]. Sampling can provide a statistical view of the program execution and help detect hotspots in the execution. The main issue we see with sampling for our analysis is that it is necessary to use the metric of interest as the sampling metric for accurate statistical results. Although the architecture of our system would allow such measurement, it effectively limits the number of metrics to collect during one run to a single metric.

B. Instrumentation-based data collection

Instead of sampling, we decided to use instrumentation-based data collection and analyze the execution at the OpenMP region level. Using instrumentation, we can collect the maximum supported number of hardware counters supported by the platform, in our case, eight counters at a time. We also need to separate OpenMP synchronization, as this is where the analysis of synchronization in the tool happens.

The first challenge when applying instrumentation-based performance analysis to dynamic data race detection is that the two tools need to run with the application simultaneously. Archer builds on the OpenMP tools interface (OMPT) to collect the OpenMP synchronization information. Most performance analysis tools commonly used in HPC like Score-P [12], TAU [13], or Paraver [14] also started to build on OMPT to collect OpenMP specific information. The OMPT interface as specified in the OpenMP specification since version 5.0 does not support multiple tools. However, if a tool is built with the header-only OMPT multiplex extension [15], the tool can load another OMPT tool allowing a nested execution of multiple tools.

In a first attempt, we tried to collect the performance data with Score-P. We instrumented the annotation calls in the Archer runtime library with user-defined regions to see the impact of happens-before annotation in Archer. Unfortunately, this instrumentation is neither compatible with OPARI-based³ nor with OMPT-based OpenMP instrumentation. Score-P does not accept user instrumentation outside of OpenMP regions, which we need for keeping track of the synchronization annotation in Archer. Instead, we collect execution traces using Score-P by completely disabling OpenMP support in Score-P and using a small OMPT tool to create user-defined regions representing OpenMP regions. The tool assigns unique names to the different OpenMP parallel and task regions as the execution encounters them. The unique names allow the creation of performance profiles for the individual regions.

³OPARI is a source to source compiler to instrument OpenMP code

C. Benchmarking framework for data race detection

The second challenge for this kind of performance analysis is the vast amount of data and how to compare the behavior across different configurations. Since we didn't find a tool supporting comparison of more than two measurements at a time among the commonly used HPC performance analysis tool, we decided to generate graphs from the profiling data using `python` and `matplotlib`. The automatically generated plots allow us to quickly compare the impact of different tool configurations to specific metrics for each of the OpenMP regions of an application.

All graphs presented in this paper are generated automatically with our benchmarking framework. For better readability in this paper, the font size is increased compared to the generated reports, where each graph is shown as a landscape page. The framework consists of a set of configuration files for the JUBE benchmarking environment [10] and a set of post-processing scripts to generate the diagrams.

The first step of the workflow is to launch the experiment with `jube run` and waiting for the batch jobs to finish. For this step, it is possible to select different sets of hardware counters for collection. The command `jube continue` ensures that all JUBE steps completed. When all JUBE steps are finished, launching a python script is sufficient to generate the different PDF reports containing all the graphs.

All configuration files, scripts, and generated plots are available on GitHub⁴. The configuration only needs a few adjustments to apply the workflow to other analysis tools on other platforms. Firstly, the compilation commands for the tool must be added and possible environmental variables to control the tool behavior can be specified. Secondly, the environment for compilation and execution must be specified which is typically done by loading some environmental modules. Finally, the system-specific configuration might need some updates to allow compiling and running the codes on a different system. For more details on the framework we refer to [16].

D. Performance analysis workflow summary

For the first overview, a diagram like in Figures 2 and 8 hints at problematic applications with the highest runtime overhead caused by the analysis tool. Starting from there, we can look at the individual application and inspect the tool overhead at a per OpenMP region level as in Figure 3. The implicit task regions in this figure correspond to different OpenMP parallel regions in the application code or in linked libraries. We identify the OpenMP region of interest, either the region with the highest relative overhead or the region with the highest absolute runtime under tool control. In case of `bwaves` shown in Figure 3, parallel region 10 (`ImplTaskRegion10`) has the highest relative runtime overhead (965 \times) and the highest execution time under tool control marked with the \times mark. For brevity we filtered regions with less than 50 \times runtime overhead and less than 10 seconds base runtime from the figure. Finally, we inspect the collected hardware performance counter for the

⁴<https://github.com/RWTH-HPC/drd-performance>

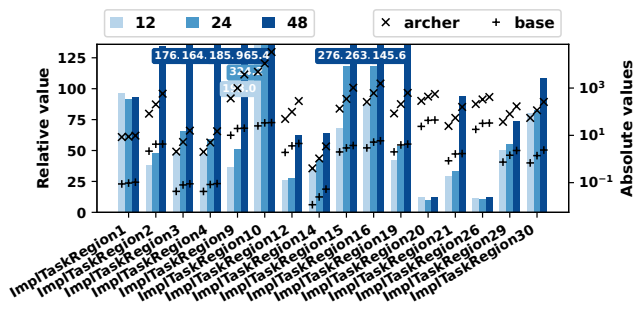


Fig. 3: Relative runtime overhead of Archer for the different parallel regions of `bwaves`, filtered for overhead > 50 \times or base > 10s.

region of interest and correlate spiking relative counter values with the observed time overhead.

For the following considerations, we already identified 351.`bwaves`, 352.`nab`, and 370.`mgrid331` as the interesting applications and also identified a parallel region of interest for each of the codes.

E. Evaluation of collected data

Neither the count of retired instructions nor the counts related to branches could explain the runtime overheads for the three applications. Figures 4-6 show the breakdown of memory accesses to the cache level where the data was found. For now, let's only consider the left three charts `base`, `tsan`, and `archer` of each figure. The more light blue we see in the chart, the more L1 cache reuse we can observe. If an application performs linear memory accesses and uses all values of a cache line, we should see 7/8 light blue assuming a cache line size of 64 byte and 8 byte (`double`) accesses.

The `base` chart for `bwaves` indicates a more or less random access pattern in this code region with no L1 cache reuse. Since ThreadSanitizer (`tsan`) in most cases will read all four shadow words corresponding to the application word and therefore reuse the cache line of the shadow memory⁵ at least

⁵Section V will give more details on the use of shadow memory

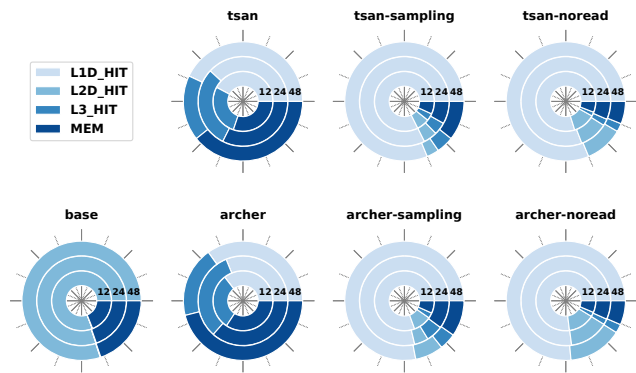


Fig. 4: Cache utilization of different tool configurations on parallel region 10 of 351.`bwaves` for 12, 24, and 48 threads.

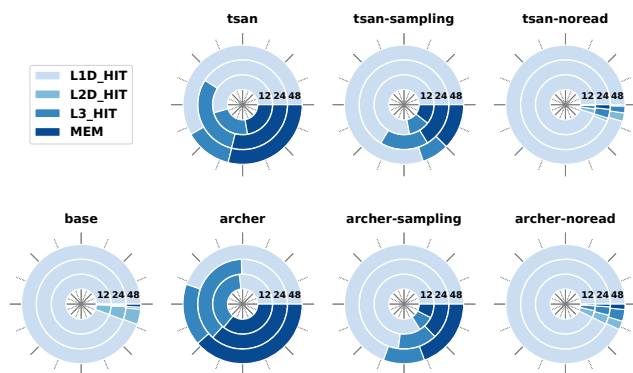


Fig. 5: Cache utilization of different tool configurations on parallel region 7 of 352.nab for 12, 24, and 48 threads.

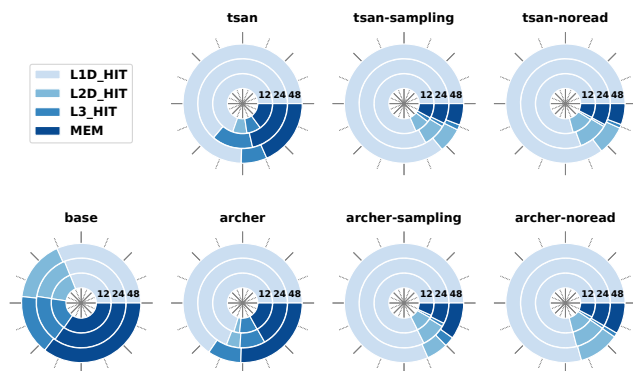


Fig. 6: Cache utilization of different tool configurations on parallel region 9 of 370.mgrid331 for 12, 24, and 48 threads.

three times, we might expect at least 75% light blue for the tool charts. However, ThreadSanitizer does not only access the directly mapped shadow memory, which explains fewer L1 cache hits for this configuration. The pie charts only show the counts of cache accesses relative to all memory accesses. The charts do not show the absolute values of memory accesses. Looking into the data, ThreadSanitizer and Archer (**archer**) actually perform 60× respectively 40× as many read accesses as the baseline. Still, the charts show a higher percentage of main memory accesses than the baseline execution, with an increasing trend for larger thread counts. This shift in cache utilization and the increased number of memory accesses can explain the increasing runtime overhead.

The **base** chart for **nab** indicates perfect L1 cache reuse and a problem size fitting into the L2 cache. In contrast, both Archer and ThreadSanitizer often need to access the main memory. Looking into absolute values, we can say that ThreadSanitizer for this region performs almost double the memory access than Archer, which explains the difference in the percentages while the absolute numbers of main memory accesses are almost the same. Again, this shift in cache utilization and the increased number of memory accesses can explain the increasing runtime overhead.

List. 1: Matrix-vector multiplication example (MxV)

```
void mxv_row(int m, int n, double *A, double *B,
            double *C) {
#pragma omp parallel for
for (int i = 0; i < m; i++)
for (int j = 0; j < n; j++)
A[i] += B[i * n + j] * C[j];
}
```

For **mgrid331**, the **base** chart shows a pretty consistent distribution of cache level accesses across the different numbers of threads, including a significant proportion of main memory accesses. Again the reuse of cache lines for shadow memory explains a higher percentage of L1 accesses for ThreadSanitizer and Archer. Looking into absolute values, ThreadSanitizer and Archer perform about 40× as many read accesses as the baseline, and they have about 14× as many main memory accesses as the baseline.

Now the question is, what is the source of these main memory accesses, and why are they so bad for the performance? Looking into the source code of the problematic parallel regions, we realize that in all cases, the different threads work on some common input data that all threads read concurrently. In the next section, we will discuss the impact of such memory access patterns for the ThreadSanitizer analysis and propose an optimization for ThreadSanitizer to overcome this performance issue.

V. RUNTIME OPTIMIZATION FOR TSAN

Based on the performance analysis from the previous section, we conclude that concurrent read accesses of a shared variable are responsible for significant runtime overheads in some of the SPEC OMP benchmarks. A simple example for such a memory access pattern is the OpenMP-parallel matrix-vector multiplication shown in Listing 1. All threads concurrently read the input vector **C** of size n , while multiplying it with the different rows of the $n \times m$ matrix **B**. Each thread writes the result to different elements of output vector **A**. So, there is neither a data race nor other concurrent write access to shared data. Considering the memory access to **C** from a cache hierarchy point of view in a NUMA system, the read-only access allows for redundant loads of the data into the local caches of the different cores.

Adding data race detection to the execution significantly changes the caching behavior. A data race detection tool needs to store information about the memory accesses to some *log file*. In the case of ThreadSanitizer, this log file is implemented as an arithmetically mapped shadow memory, i.e., the mapping between application and shadow memory addresses is a linear function. For each 64 bit word of shared application memory, ThreadSanitizer maintains four words of shadow memory to store information about the latest memory accesses to the application word. After some filtering of memory accesses and comparing the memory access with the four shadow entries for data race, ThreadSanitizer stores the information about the memory access to one of the four shadow words, unless the

data race detection phase already identified an entry suitable for replacement. Although ThreadSanitizer implements all this lock-free, the write access invalidates the local copies of the cache line for all other cores, enforcing them to fetch a new copy from memory before logging their access to the same variable. Effectively, the data race analysis transforms the harmless shared read memory access pattern into an expensive shared write memory access pattern.

A. Sampling of concurrent read accesses

To highlight the performance impact of the random replacement of shadow entries, we modified ThreadSanitizer to skip the logging of read accesses if they were not already logged during the data race detection phase. As Figure 7 shows, this strategy can reduce the runtime overhead from 2800x (**main**) to 30x (**noreads**) for execution with 48 threads. Similarly, the **noreads** charts in Figures 4-6 highlight the impact of dropping such concurrent read accesses from logging.

To not lose the information for all read accesses, we propose a sampling strategy for the read accesses to be randomly logged. As runtime configuration value, we use a **sampling_level** s to specify the sampling rate 2^s , which means that only every 2^s th read access that would be logged after data race detection is actually stored. All other read accesses are still analyzed for data race but are not logged for comparison with subsequent memory accesses.

B. Possible impact of sampling to accuracy

Even with skipping the logging of read accesses, we can detect most data races if we spot the read access after a concurrent write access.

Following the reasoning of FastTrack [17], ThreadSanitizer only stores the latest write access for a memory location. All other write accesses to the location are synchronized with this write access, or otherwise, ThreadSanitizer would already have detected a data race between the two write accesses. Similarly, ThreadSanitizer overwrites an old read access log if the two read accesses are synchronized, especially for accesses from the same thread. Since ThreadSanitizer uses four shadow words per application word, multiple of these shadow words might only be filled with information about write accesses if the granularity of application memory accesses is smaller than word size. For most HPC applications we expect data types of at least four bytes (**int/float**) and often even word size (**double**). The remaining shadow words will typically be filled with read accesses.

For applications with rare write accesses to a shared variable and many read accesses to the same variable, **noreads** and **sampling** can even increase the chance to detect a data race. For such a memory access pattern in the current implementation, synchronized read accesses might overwrite the shadow entry of the write access without having a data race, while subsequent unsynchronized read accesses will not be detected as race.

Suppose granularity of memory accesses by the application is smaller than half-word, such as half-precision float in

List. 2: Code example with data race between line 13 and 18. where **noread** will miss the data race while **main** will detect the data race. If line 7 is commented in, **main** will miss the data race while **noread** will detect the data race

```
1   int a = 1, sum = 0, b = 0;
2   #pragma omp parallel num_threads(2)
3   {
4       if(omp_get_thread_num()==0)
5       #pragma omp parallel num_threads(5) reduction(+:sum)
6       {
7           // usleep(100000);
8           for (int i=0; i<10; i++)
9               sum += a; // fill shadow for a
10          usleep(100000);
11          #pragma omp barrier
12          #pragma omp critical
13          a++; // racy writes
14      }
15      if(omp_get_thread_num()==1)
16      {
17          usleep(50000);
18          b = a; // racy read
19      }
20  }
```

machine learning applications. In that case, all shadow words could eventually be filled with write accesses, and for **noreads** only write accesses will be able to update the shadow state.

To understand this weakness but also the strength of the **noreads** and **sampling** implementation we consider the memory accesses to **a** in Listing 2 where we use sleep to emulate different workload between the threads to enforce some time interleaving. In the first line, the initial thread writes to **a** creating a write entry in the first shadow cell. The threads spawned for the outer parallel region are synchronized with this write access. The first thread of the outer parallel region spawns a second inner parallel region. All threads in this region read **a**, filling the remaining three shadow cells with information about the read accesses. The sleep in the second outer thread makes sure that the shadow cells are filled before the read access of **a** by this thread. This read access is not logged during the data race detection phase, as it is concurrent to all memory accesses in the shadow cells. Finally, all threads in the inner parallel region increment **a** under mutual exclusion provided by the **critical** construct causing a data race with the read access by the second outer thread. The unmodified ThreadSanitizer detects the data race and will even print a warning: **As if synchronized via sleep**

Now we add the sleep in line 7. The racy read by the second outer thread executes first creating an entry in shadow memory. With the **noreads** and **sampling** configuration the read access is not overwritten—or only with low chance—allowing this configuration to detect the data race. The unmodified ThreadSanitizer has no chance to detect the data race, because the read accesses overwrite the log entry from the second outer thread. Starting with a sampling level of 4 the data race is detected for this kernel.

If there is just a single race in an application, the **noreads**

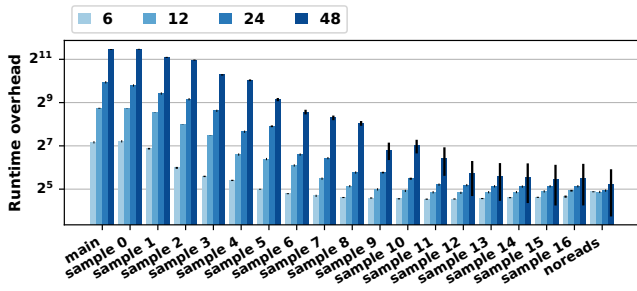


Fig. 7: Overhead of Archer with different sampling levels for MxV kernel in Listing 1 ranging from 30x to 3000x

and **sampling** approach have the chance that for specific ordering of the execution the race can be detected when the read occurs after the write. The **noreads** configuration guarantees to detect all read after write data race. The **sampling** configuration increases the chance to detect certain write after read data race while maintaining also a good chance to detect most read after write data races.

VI. PERFORMANCE RESULTS

Figure 7 highlights the impact of different sampling levels on the runtime overhead of Archer and ThreadSanitizer analysis for contentious shared read accesses. As explained before, increasing the sampling level by one doubles the number of dropped read accesses. We choose the probably unusual logarithmic y-axis with base 2 because it supports reading the doubling of runtime.

Due to the simple nature of the kernel, clang would optimize away the contentious read accesses under higher optimization levels, so we decided to disable code optimization when compiling the **mxv** kernel function. To confirm that our experimental toolchain using gfortran for compilation and clang for execution is not the origin of the significant performance difference, we compared the behavior with an equivalent Fortran implementation of the **mxv** kernel function. The observed overhead for the Fortran version is consistent with the overhead shown in Figure 7.

An important observation in this figure is that **sample 0** shows the same overhead as **main** which demonstrates that our sampling implementation does not introduce overhead when disabled.

A. Performance impact of sampling for SPEC OMP 2012

The charts **tsan-sampling** and **archer-sampling** in Figures 4-6 show the impact of sampling level 8 to the cache utilization of the analyzed parallel regions. Especially for **nab** in Figure 5 this sampling level does by far not reach the possible performance gain demonstrated by the **noreads** configuration. Figure 8 compares the runtime overhead of ThreadSanitizer and Archer for the configurations **main**, **sampling**, and **noreads**. Again, the **sampling** configuration uses sampling level 8, i.e., samples only every 256th read access that was not logged during the data race analysis step.

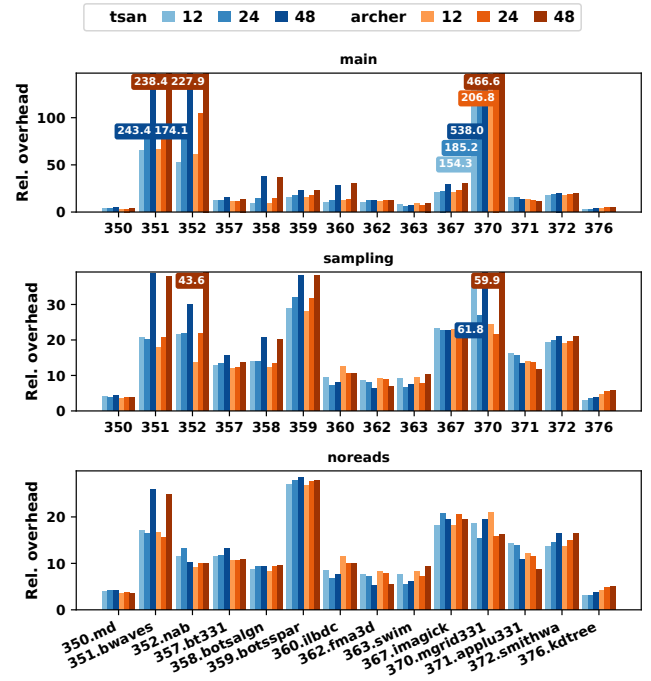


Fig. 8: Relative runtime overhead of ThreadSanitizer or Archer for SPEC OMP 2012 with **drd** input data and 12, 24, and 48 threads.

As the figure shows, this sampling level is sufficient to cut down the runtime overhead introduced by the bookkeeping of memory accesses in ThreadSanitizer and Archer from about $500\times$ to about $60\times$ for **mgrid** and from around $200\times$ to about $40\times$ for **bwaves** and **nab**. Further experiments showed that a sampling level of 10 is sufficient to limit the runtime overhead to $40\times$ for all benchmarks. The apparent overhead increase for **botsspar** is just an optical illusion caused by the change of scaling in the y-axis.

VII. CONCLUSIONS

In this paper, we proposed the novel input data set **drd** for the SPEC OMP 2012 benchmark. The input data is designed to execute about 10 seconds with 12 threads on current CPUs leaving room to observe even significant runtime overheads of $500\times$ introduced by dynamic data race detection tools without running into timeout issues.

Furthermore, we presented a performance analysis workflow for data race detection using the JUBE benchmarking environment. The configuration only needs a few adjustments to apply the workflow to other analysis tools on other platforms. Following the performance analysis workflow, we identified contentious read accesses to shared data as the root cause of severe performance issues for certain scientific applications.

As a third contribution, we presented a heuristic optimization to sample contentious read accesses. We showed that this optimization could cut the analysis overhead to an upper limit of $60\times$ with a sampling level of 8. We assess that a sampling level of 10 cuts down all overheads for the SPEC OMP 2012

benchmark to below $40\times$. A benchmark to evaluate the impact of sampling to the detection rate is subject to future work.

ACKNOWLEDGEMENTS

This work has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement 824080. We gracefully thank the reviewers for their valuable feedback.

REFERENCES

- [1] Y. Gu and J. M. Mellor-Crummey, “Dynamic data race detection for openmp programs,” in *SC*. IEEE / ACM, 2018, pp. 61:1–61:12.
- [2] J. Protze *et al.*, “Towards providing low-overhead data race detection for large openmp applications,” in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014*, 2014, pp. 40–47.
- [3] U. Bora *et al.*, “LLOV: A fast static data-race checker for openmp programs,” *CoRR*, vol. abs/1912.12189, 2019.
- [4] S. Atzeni *et al.*, “SWORD: A bounded memory-overhead detector of openmp data races in production runs,” in *IPDPS*. IEEE Computer Society, 2018, pp. 845–854.
- [5] A. J. Dorta, C. Rodríguez, F. de Sande, and A. González-Escribano, “The OpenMP Source Code Repository,” in *PDP*. IEEE Computer Society, 2005, pp. 244–250.
- [6] C. Liao *et al.*, “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*, 2017, pp. 1–14.
- [7] S. Atzeni *et al.*, “ARCHER: Effectively Spotting Data Races in Large OpenMP Applications,” in *IPDPS*. IEEE Computer Society, 2016, pp. 53–62.
- [8] M. S. Müller *et al.*, “SPEC OMP2012 - An Application Benchmark Suite for Parallel Systems Using OpenMP,” in *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP*, 2012, pp. 223–236.
- [9] J. Protze *et al.*, “OpenMP Tools Interface: Synchronization Information for Data Race Detection,” in *Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP*, 2017, pp. 249–265.
- [10] S. Lührs *et al.*, “Flexible and Generic Workflow Management,” in *Parallel Computing: On the Road to Exascale*, ser. Advances in parallel computing, vol. 27. International Conference on Parallel Computing 2015, Sep 2016, pp. 431–438.
- [11] L. Adhianto *et al.*, “HPCTOOLKIT: tools for performance analysis of optimized parallel programs,” *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 685–701, 2010.
- [12] C. Feld *et al.*, “Score-P and OMPT: Navigating the Perils of Callback-Driven Parallel Runtime Introspection,” in *IWOMP*, ser. Lecture Notes in Computer Science, vol. 11718. Springer, 2019, pp. 21–35.
- [13] A. D. Malony *et al.*, “A plugin architecture for the TAU performance system,” in *ICPP*. ACM, 2019, pp. 90:1–90:11.
- [14] F. Mantovani and E. Calore, “Multi-node advanced performance and power analysis with paraver,” in *PARCO*, ser. Advances in Parallel Computing, vol. 32. IOS Press, 2017, pp. 723–732.
- [15] J. Protze, T. Cramer, S. Convent, and M. S. Müller, “OMPT-Multiplex: Nesting of OMPT Tools,” ser. SpringerLink. Cham: Springer, Sep 2019, pp. 73–83.
- [16] I. Thäringen, J. Protze, F. Orland, and M.-A. Hermans, “Differential Performance Analysis Workflow for Algorithmic Changes,” in *Pro-Tools@SC*. IEEE, 2021.
- [17] 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 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 121–133.

APPENDIX

For the application *360.ilbdc* the **test** input is reused for **drd**. For the application *362.fma3d* the **train** input is reused for **drd**.

For the applications *352.nab* (see Listing 5), *359.botsspar* (see Listing 8), *367.imagick* (see Listing 10), *370.mg* (see Listing 11), *372.smithwa* (see Listing 13), and *376.kdtree* (see Listing 14) the **drd** input file is given in the listings.

The **drd** input for the applications *350.md* (see Listing 3), *351.bwaves* (see Listing 4), *357.bt331* (see Listing 6), *358.botsalgn* (see Listing 7), *363.swim* (see Listing 9), and *371.applu331* (see Listing 9) is derived from **test**, **train**, or **ref** input as described in the captions by changing some lines.

List. 3: *350.md/data/drd/input/runmd.in* derived from **train** data by changing the following lines:

```
13  nind = 6, !number of steps between
    ↪ measurements
24  8544, 6.00, 12.00,
25  1824, 8.00, 16.00,
```

List. 4: *351.bwaves/data/drd/input/bwaves.in* derived from **train** data by changing the following line:

```
4  144 144 144
```

List. 5: *352.nab/data/drd/input/control*:

```
1  gcn4p1 1850041461
```

List. 6: *357.bt331/data/drd/input/inputbt.data* derived from **test** data by changing the following line:

```
3  80 80 80
```

List. 7: *358.botsalgn/data/drd/input/botsalgn* derived from **ref** data by changing the following line and cutting off after line 401:

```
1  Number of sequences is 200
```

List. 8: *359.botsspar/data/drd/input/control*:

```
1  botsspar 120 90
```

List. 9: *363.swim/data/drd/input/swim.in* derived from **train** data by changing the following lines:

```
8  1734
9  1734
```

List. 10: *367.imagick/data/drd/input/control*:

```
1  # Format is:
2  # exename outputfile errfile time? arg [arg...]
3  #
4  convert convert1.out convert1.err 1 -shear 31 -resize
  ↪ 1280x960 -negate -edge 14 -implode 1.2 -flop -convolve
  ↪ 1,2,1,4,3,4,1,2,1 -edge 100 input1.tga output1.tga
```

List. 11: *370.mg/data/drd/input/mg.input*:

```
1  9 = top level
2  256 128 1024 = nx ny nz
3  150 = nit
4  0 0 0 0 0 0 0 = debug_vec
```

List. 12: *371.applu331/data/drd/input/inputlu.data* derived from **train** data by changing the following line

```
18 90 90 90
```

List. 13: *372.smithwa/data/drd/input/control*:

```
1  testset 35
```

List. 14: *376.kdtree/data/drd/input/control*:

```
1  testset 350000 10 2
```