Towards Providing Low-Overhead Data Race Detection for Large OpenMP Applications*

Joachim Protze†, Simone Atzeni*, Dong H. Ahn‡,
Martin Schulz‡, Ganesh Gopalakrishnan*, Matthias S. Müller†,
Ignacio Laguna‡, Zvonimir Rakamarić*, Greg L. Lee‡
†RWTH Aachen University *University of Utah
‡Lawrence Livermore National Laboratory
†{protze, mueller}@itc.rwth-aachen.de
*{simone, ganesh, zvonimir}@cs.utah.edu
‡{ahn1, schulzm, lagunaperalt1, lee218}@llnl.gov

ABSTRACT

Neither static nor dynamic data race detection methods, by themselves, have proven to be sufficient for large HPC applications, as they often result in high runtime overheads and/or low race-checking accuracy. While combined static and dynamic approaches can fare better, creating such combinations, in practice, requires attention to many details. Specifically, existing state-of-the-art dynamic race detectors are aimed at low-level threading models, and cannot handle high-level models such as OpenMP. Further, they do not provide mechanisms by which static analysis methods can target selected regions of code with sufficient precision. In this paper, we present our solutions to both challenges. Specifically, we identify patterns within OpenMP runtimes that tend to mislead existing dynamic race checkers and provide mechanisms that help establish an explicit *happens-before* relation to prevent such misleading checks. We also implement a fine-grained blacklist mechanism to allow a runtime analyzer to exclude regions of code at line number granularity. We support race checking by adapting ThreadSanitizer, a mature data-race checker developed at Google that is now an integral part of Clang and GCC; and we have implemented our techniques within the state-of-the-art Intel OpenMP Runtime. Our results demonstrate that these techniques can significantly improve runtime analysis accuracy and overhead in the context of data race checking of OpenMP applications.

1. INTRODUCTION

OpenMP is the de facto standard for parallel programming on shared memory machines. It is also becoming increasingly popular on extreme-scale systems as it offers a portable way to harness the growing degree of parallelism available on each node. However, porting large HPC applications to OpenMP often introduces subtle errors. Of these, data races [4] are particularly egregious, as well as challenging to identify. Data races may remain undetected during testing, but nevertheless manifest during production runs by often resulting in confusing (and/or non-reproducible) executions that the programmer wastes considerable amounts of time debugging. In extreme situations, data races may simply end up silently corrupting user data. For all these reasons, data race detection remains one of the central concerns in parallel programming, in particular for shared memory programming models.

While considerable progress has been made in data race detection applied to lower-level threading models, very few practical tools are available for higher-level threading models, like OpenMP. For example, while mature runtime tools are readily available, which includes Helgrind [8] and ThreadSanitizer [16], they primarily aim at POSIX threads and often do not support high-level programming models built on top of them.

The tools that do exist for OpenMP are based on either pure static or dynamic analysis techniques. While effective on small- to medium-sized applications, these approaches fall short when analyzing large HPC applications that are characterized by high memory usage, complex interplay with other types of parallelism, and non-trivial code sizes. While runtime approaches are often highly accurate, even modern runtime techniques can incur an over 30 to 100 fold execution slowdown and over a factor-of-10 memory overhead. Often these overheads are unacceptably high to diagnose HPC applications that can run for several days and use large portions of system memory. In contrast, while static analysis techniques incur low runtime overheads, they are imprecise by nature and thus can produce many false alarms.

It is well known that combined static and dynamic approaches can help to improve data-race checking significantly without sacrificing analysis accuracy and precision. For instance, as part of code compilation, such approaches can apply a set of static analysis passes to classify the code into race-free and potentially racy regions. However, to the best of our knowledge, such a combination has not been attempted for OpenMP programs — especially real applications that are meant to run on state-of-the-art OpenMP runtimes. In this paper, we show that any attempt to transition to practice with these objectives immediately introduces many technical challenges. We identify the following two primary challenges: (1) *a programming model challenge,*

*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-660004).

978-1-4799-7023-0/14 $31.00 © 2014 IEEE
DOI 10.1109/LLVM-HPC.2014.7

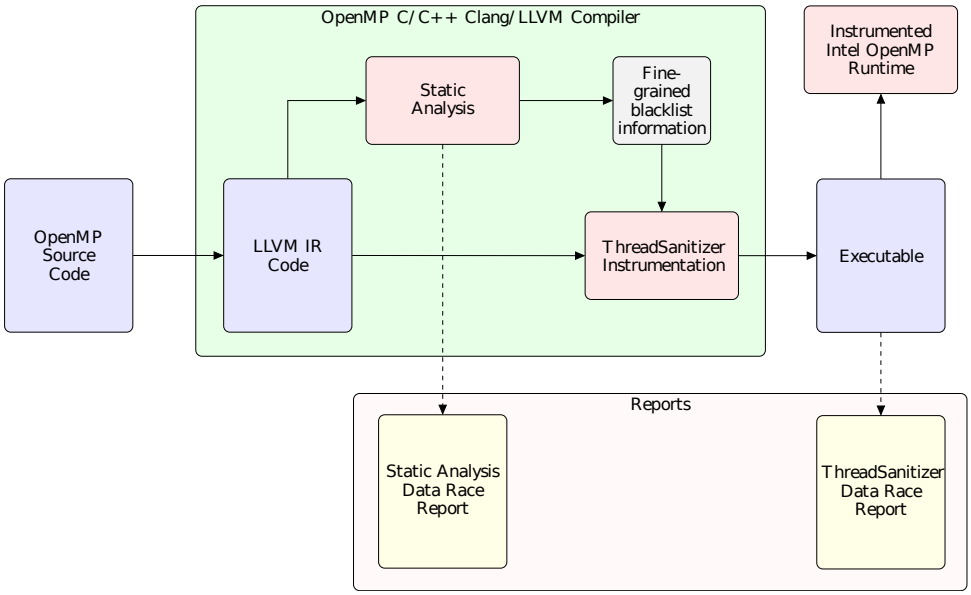


Figure 1: Overall approach of Archer

which refers to the need to handle the higher-level primitives of OpenMP, and (2) *a static-dynamic collaboration challenge*, which refers to the fact that available dynamic data-race detection tools do not come with mechanisms by which they can target selected regions of a given piece of code to be analyzed. In this paper, we present our solutions to both challenges.

To address the programming model challenge, we demonstrate how to identify situations in which low-level synchronization patterns within implementations of concurrency libraries can mislead runtime techniques. This occurs because of the mismatch between the happens-before relation assumed by these runtime methods and those embedded within the libraries. Our solution is to build the knowledge of this happens-before relation into the runtime libraries to obtain race-checking precision.

To address the static-dynamic collaboration challenge, we use a fine-grained blacklist mechanism whereby a runtime analysis can exclude regions of code for checking at line-number granularity. While the general use of blacklisting enhances the speed of checking at the expense of completeness, in some cases we can avoid losing completeness because of an LLVM-level static analysis pass that is capable of identifying guaranteed race-free regions that do not need to be instrumented.

In summary, we make the following contributions:

- A blueprint of a low-overhead OpenMP data-race checker;
- The identification of thread synchronization patterns within a widely used OpenMP runtime, which often misleads runtime checkers with false alarms;
- Instrumentation techniques that can eliminate these false alarms; and
- A novel blacklist technique that allows runtime checkers to select and examine targeted code regions at a fine granularity.

We implemented our techniques in the Intel OpenMP Runtime [1], a widely used OpenMP runtime library, and ThreadSanitizer [16] (TSan), a popular open-source runtime data-race checker. Our evaluation on OmpSCR, an OpenMP source code collection, shows that the instrumented OpenMP library allows TSan to eliminate all false alarms without decreasing data-race analysis accuracy and precision. Our evaluation also shows that the fine-grained blacklist technique sufficiently allows TSan to exclude its runtime analysis at the OpenMP region boundary. Further, both techniques significantly reduce the runtime performance and memory overheads of TSan.

2. APPROACH

To provide an effective and scalable analysis of data races in large OpenMP applications, we must bring the best from both static and dynamic analysis techniques and combine them in a seamless workflow: as part of the compilation process, we first apply a set of static analyses, such as loop-carried data dependency and thread-escape analysis, to classify OpenMP code into race-free, certainly racy or potentially racy regions and pass only the potentially unsafe regions into a runtime analyzer.

We build on top of the OpenMP branch [2, 3] of LLVM/Clang [11, 12] and ThreadSanitizer (TSan) [16] and started to prototype a tool called Archer. Our tool composes static analyses with dynamic techniques to create a seamless analysis workflow, as illustrated in Figure 1. On the static side, Archer builds on the Clang/LLVM suite and utilizes some of the static and dynamic verification passes already present in LLVM. Currently, it uses Polly [7], the data-dependency analysis and loop-carried data-dependency analysis in LLVM. Specifically, once the OpenMP code is translated into the 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 example OpenMP code where our static data dependency analysis can guarantee data-race freedom: with no data dependency, threads *will* each exclusively

```
#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 dependencies

Figure 2: OpenMP parallel for loop

access distinct array locations. On Figure 2b, applying data-dependency analysis discovers that this loop has a loop-carried data dependency. This means 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 will be further analyzed at execution time for more accurate diagnosis.

On the runtime side, we implement Archer by extending TSan to include a blacklist feature so that it can exclude code from being examined during runtime at finer granularity (e.g., at the line-number or OpenMP-region level) than its current version. Further, because TSan does not currently support OpenMP, we also instrumented the Intel OpenMP Runtime to allow TSan to recognize its synchronization primitives.

3. THREADSANITIZER

The core of the TSan runtime is a state machine based on three basic patterns [13, 14]: memory access (READ, WRITE), synchronization (SIGNAL, WAIT) and locking (LOCK, UNLOCK).
The access to memory is logged into a shadow memory. The virtual memory address space is partitioned into a segment for application memory and a segment for shadow memory, each embedded in restricted memory regions. The mapping between application memory and shadow memory is a simple algebraic expression. A log entry includes the current program counter and the thread id.
The synchronization information is logged in a vector of scalar clocks containing the remote program counter from the last synchronization with the remote thread. The basic synchronization pattern in TSan is sending a signal from one thread and waiting for this signal at another thread. This kind of synchronization is highlighted by a `happens-before arc` starting at the signal call and pointing to the wait call.

Annotation API. The Annotation API of ThreadSanitizer provides API functions to highlight memory usage and synchronization. In particular, as discussed in the next section, we use the API to make TSan understand the OpenMP synchronization semantics by identifying the respective synchronization points:

- `AnnotateRWLockAcquired(char *f, int l, uptr mem)`
- `AnnotateRWLockReleased(char *f, int l, uptr mem)`
- `AnnotateHappensBefore(char *f, int l, uptr mem)`
- `AnnotateHappensAfter(char *f, int l, uptr mem)`
- `AnnotateIgnoreWritesBegin(char *f, int l)`
- `AnnotateIgnoreWritesEnd(char *f, int l)`

Most of the API functions rely on the shadow memory to implement their functionality. The memory addressed by the `mem` parameter is used to store synchronization information as thread id and local clock value. As a result this address should be unique for the pair of threads or the used lock and the address should not be used in other application context. At the same time, this address argument needs to be known by both of the synchronizing threads, so it needs to be shared knowledge between the threads. The Read-Write-Lock functions may be used to highlight user defined locks. The Happens-Before/After functions highlight the start and respectively end of a happens-before markup. Finally the Ignore-Write functions mark a region of code where writes to memory should be ignored by the runtime analysis.

4. SOLUTIONS

As a foundation to our proposed approach described in Section 2, a strong body of work exists in the fields of both static and dynamic analysis techniques. However, we find two important gaps can impede our effort to blend these two classes of techniques.

First, while mature state-of-the-art runtime tools exist, including Helgrind and ThreadSanitizer, they aim primarily at low-level POSIX/OS-level threading paradigms and often do not support high-level programming models like OpenMP. Specifically, they do not recognize raw synchronization primitives and/or patterns that OpenMP runtime implementations use, and as a result report an excessive number of false alarms on large production OpenMP applications. Further, these runtime tools do not provide a mechanism by which the static counterpart can target certain regions of code at a sufficiently fine granularity.

In the following, we present our solution to both challenges. While our solution is proposed and validated within the Archer workflow that uses LLVM and ThreadSanitizer and targets OpenMP, it applies, in principle, to any static-dynamic approach to data-race detection and to any high-level programming model with threading semantics.

4.1 Highlighting Synchronization

The OpenMP standard specifies several high-level synchronization points. Explicit synchronization points include `#pragma omp barrier`, `#pragma omp critical`, and `#pragma omp taskwait`. Implicit synchronization include `#pragma omp single`, `#pragma omp task`, and the OpenMP `reduction` clause. As semantically intended and realized in the Intel OpenMP runtime, the threads can enter a critical section, as the one shown in Figure 3a, only in a serialized manner, avoiding a data race in this example. Lacking the knowledge about these synchronization points, however, ThreadSanitizer generates false alarms, as shown in Figure 3b. We use the Annotation API of TSan to highlight these synchronization points to avoid such false positives.

```
#include <stdio.h>
int main(int argc, char **argv)
  int a=0;
  #pragma omp parallel
  {
    #pragma omp critical
    {
      a++;
    }
  }
  printf("Sum: %i\n", a);
}
```

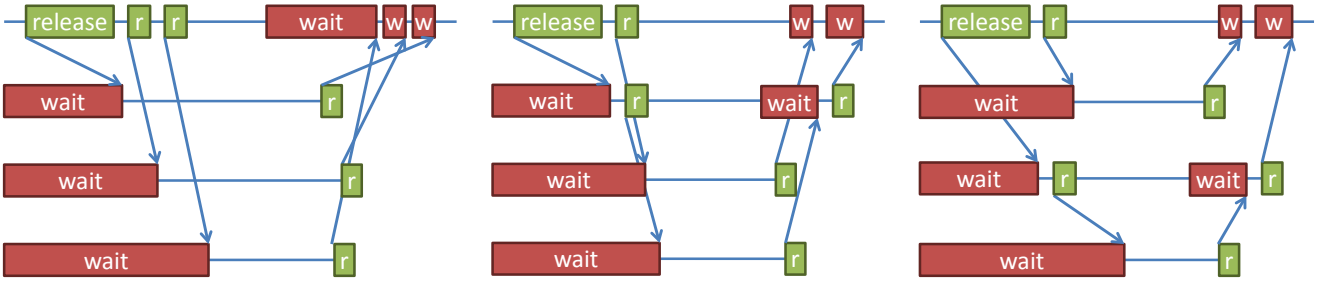
(a) Race-free OpenMP critical section example

```
WARNING: ThreadSanitizer: data race (pid=49285)
  Write of size 4 at 0x7fd8dc by thread T2:
    #0 .omp_microtask. critical.c:8
       (critical+0x0000000a86f5)
    #1 __kmp_invoke_microtask <null>:0
       (libiomp5.so+0x0000000b96d2)

  Previous write of size 4 at 0x7fd8dc by thread T4:
    #0 .omp_microtask. critical.c:8
       (critical+0x0000000a86f5)
    #1 __kmp_invoke_microtask <null>:0
       (libiomp5.so+0x0000000b96d2)
```

(b) ThreadSanitizer report for OpenMP critical section example

Figure 3: False alarm message example for uninstrumented Intel OpenMP Runtime



(a) Linear Barrier **(b)** Tree Barrier **(c)** Hypercubic Barrier

Figure 4: The blue arrows represent happens-before relations for fork/join barrier implementations in the Intel OpenMP Runtime.

Locking Algorithms. The Intel OpenMP Runtime implements several low-level locking mechanisms for initialization purposes. By default, the locking algorithm in use is the Lamport Bakery Algorithm [10] , which is a queuing lock algorithm. The Lamport Bakery Algorithm works for situations with unclear knowledge of concurrent threads. To give some examples, this kind of lock is used to protect the initialization of Pthread locks and to protect some sections of the fork and join procedures. The queuing lock is used to implement the locks for OpenMP pragmas like `critical` or `lock`. Using the ThreadSanitizer annotation functions `AnnotateRWLockAcquired` and `AnnotateRWLockReleased` we were able to resolve the false alarms for high-level OpenMP locking pragmas like `#pragma omp critical`.

Fork/Join Barrier. We now explain how we conveyed information about the happens-before relation pertaining to the barriers used in the Intel OpenMP Runtime. Since the origins of the OpenMP programming model is based on a fork-join model, most of the implicit and explicit barriers underpin the synchronization for fork, join or join-fork operations. The Intel OpenMP Runtime implements 3 variants for the fork and the join barrier:

- Linear barrier: one thread (master) synchronizes with all others,
- Tree barrier: synchronization is implemented along a binary tree, and
- Hypercubic barrier: synchronization is implemented in the dimensions of a hypercube.

The Figures 4(a-c) present these barriers along with arrows that represent the happens-before relations between the involved threads. Each of the figures shows a fork barrier on the left side and a join barrier on the right side. An explicit barrier is built by a combination of a join and a fork barrier. We use the API functions `AnnotateHappensBefore` and `AnnotateHappensAfter` to highlight the happens-before relations as shown in the figures.

Lock Initialization. We now explain our use of annotations to suppress races pertaining to certain types of lock initializations. After we highlighted the synchronization by OpenMP barriers and locks to ThreadSanitizer, we were able to focus on actual data races. We discussed the race condition listed in Listing 1, as implemented in the OpenMP Runtime library with Intel developers. If two threads approach the `if` statement concurrently, both might enter the block and initialize the mutex. According to the documentation, the attempt to initialize an already initialized mutex results in undefined behavior. In fact, there is another synchronization point that guarantees that the initialization happens on a single, well-defined thread. To mark this race as benign, we instrument the runtime as in Listing 2.

Listing 1: Possible data race for pthread_mutex_init

```
if ( th->th.th_suspend_init_count <= __kmp_fork_count )
{
  /* this means we haven't initialized the suspension
       pthread objects for this thread in this instance
       of the process */
  pthread_cond_init( &th->th.th_suspend_cv.c_cond,
       &__kmp_suspend_cond_attr );
  pthread_mutex_init( &th->th.th_suspend_mx.m_mutex,
       &__kmp_suspend_mutex_attr );
```

```
  *(volatile int*)&th->th.th_suspend_init_count =
       __kmp_fork_count + 1;
}
```

Listing 2: Ignore the write at pthread_mutex_init

```
if ( th->th.th_suspend_init_count <= __kmp_fork_count )
{
  /* this means we haven't initialized the suspension
       pthread objects for this thread in this instance
       of the process */
  AnnotateIgnoreWritesBegin(__FILE__, __LINE__);
  pthread_cond_init( &th->th.th_suspend_cv.c_cond,
       &__kmp_suspend_cond_attr );
  pthread_mutex_init( &th->th.th_suspend_mx.m_mutex,
       &__kmp_suspend_mutex_attr );
  AnnotateIgnoreWritesEnd(__FILE__, __LINE__);
  *(volatile int*)&th->th.th_suspend_init_count =
       __kmp_fork_count + 1;
}
```

Alternatively we could announce a happens-before relation between the initialization of the mutex and later attempts to initialize as in Listing 3.

Listing 3: Ignore the write at pthread_mutex_init

```
AnnotateHappensAfter(__FILE__, __LINE__,
     &th->th.th_suspend_init_count);
if ( th->th.th_suspend_init_count <= __kmp_fork_count )
{
  /* this means we haven't initialized the suspension
       pthread objects for this thread in this instance
       of the process */
  pthread_cond_init( &th->th.th_suspend_cv.c_cond,
       &__kmp_suspend_cond_attr );
  pthread_mutex_init( &th->th.th_suspend_mx.m_mutex,
       &__kmp_suspend_mutex_attr );
  *(volatile int*)&th->th.th_suspend_init_count =
       __kmp_fork_count + 1;
  AnnotateHappensBefore(__FILE__, __LINE__,
       &th->th.th_suspend_init_count);
}
```

4.2 Fine-grained Blacklists

Programmers often need to analyze only small portions or particular regions of the source code that could be affected by data races. For example, when new features in an existing code are implemented there is potentially no need to run the analysis on the entire program again, but only on the new sections.

For this purpose, ThreadSanitizer [15] provides a feature that allows developers to exclude (blacklist) particular functions or whole source files from the analysis, meaning that it is possible to select only interesting portions of code to be analyzed by the tool. The blacklist is specified manually by the developer; Figure 5 shows an example of blacklisting functions and source files.

```
# Turn off checks for a particular function
fun:MyFoo
# Turn off checks for a particular file
src:bad_source.c
```

Figure 5: Blacklisting functions and source files

When we exclude a source file or a specified function, TSan at compilation time does not instrument the specified code region, and consequently no analysis will be performed on it at runtime.

The blacklist feature of TSan is a coarse-grained selection since the smallest selectable region of code is a function. In general, programs, and especially HPC programs, often have large functions containing several different parallel constructs and loops. In programming paradigms like OpenMP, most of the code is typically sequential, with only a small portion actually being executed by multiple threads. Further, the static analysis passes of Archer will wish to focus only on some code segments within a function—for example, if the static techniques found that there are race-free parts or identified a particular code region as potentially racy. In such situations, the existing blacklist feature does not allow ignoring only parts of a function, requiring the whole function to be instrumented and thereby incurring large runtime overhead with no benefit.

```
# Blacklisting line 42 of source file
# "mysource.c" in directory
# "/path/to/source/file"
line:42,mysource.c,/path/to/source/file
```

Figure 6: Blacklisting a single line

```
# Blacklisting lines from 38 to 42 of
# directory "/path/to/source/file"
# source file "mysource.c" in
line:38-42,mysource.c,/path/to/source/file
```

Figure 7: Blacklisting a range of lines

We extend the ThreadSanitizer blacklist feature so that it can exclude code from being examined during execution time at finer granularity, more specifically at the line-number or OpenMP-region level. The existing blacklist feature [15] provides keywords *src* and *fun* for specifying a source file and a function name to ignore, respectively. We introduce a new keyword *line* to indicate the lines of code to blacklist. In particular, it is possible to specify a single line as showed in Figure 6 or a range of lines as in Figure 7. In addition to a single line or range, a programmer needs to specify a source file and its absolute path. In the case of Archer, this blacklist information is automatically generated. As mentioned in Section 2, Archer applies static techniques to classify threaded code into race-free regions; certainly racy regions; and potentially racy regions. Archer then uses the source lines of race-free regions and certainly racy regions to create the blacklist to ignore all those portions of the code that do not need further analysis at runtime.

Section 5 shows the performance benefits of our fine-grained blacklist feature.

Implementation details. We have integrated our fine-grained blacklist into TSan by seamlessly extending its existing feature.The existing feature is invoked at the compile line via a specific flag. For example, the blacklist flag as shown in Figure 8 allows to specify the file that lists the function and source file names to be excluded from the runtime analysis. The fine-grained blacklist mechanism follows the same flow. As described

in Figure 6 and 7, the programmer can now specify in this file the source lines to be excluded.

```
clang -g -O0 -fopenmp \
-fsanitize=thread \
-fsanitize-blacklist=blacklist_filename.txt \
myprogram.c
```

Figure 8: ThreadSanitizer blacklist invocation

The internal mechanism of the compiler still remains the same. The file that contains the blacklist source lines is parsed by LLVM, which then stores the information on the blacklist functions and source files in an internal data structure. Following this original scheme, we introduce a new data structure to keep the information about the blacklist source lines. The parsing algorithm is now able to understand the new keyword *line* and fill the data structure with all the necessary information. This information is then used during the LLVM IR code-generation phase: Clang simply embeds the blacklist source-line information as metadata in the generated LLVM IR instruction stream. Next, all of the requested LLVM passes start to run on this instruction stream, and one of these passes is the TSan pass. We have also modified the TSan pass to understand the new metadata information and instruments only those load and store instructions that have not been annotated to be excluded. As a result, when this instrumented program is executed all the loads and stores blacklisted at compile time will not be analyzed by the TSan runtime.

5. RESULTS

We demonstrate our improvements in data-race detection accuracy on benchmarks from OmpSCR [5], an OpenMP source code collection. With the modified Intel OpenMP Runtime, we were able to detect data races in several applications, some of which had previously been documented. We verified that all of the detected data races are actually data races. This means that our technique reported no false alarms. On the other hand, the tool did not miss any data races reported by other tools. More specifically, we were able to detect data races in `c_jacobi3`, `c_loopA.badSolution`, `c_loopB.badSolution1`, `c_loopB.badSolution2`, `c_md`, `c_testPath`, `cpp_qsomp1`, `cpp_qsomp2`, `cpp_qsomp3`, `cpp_qsomp4`, and `cpp_qsomp6`. The data race in `c_md` is due to an out-of-bound access introduced by a faulty loop index variable. And this race manifests itself only for a problem size of 4096 particles, which makes this bug highly elusive.

In Figure 9, we present our overhead measurements for these benchmarks. We compare the execution times of the application kernels at 2 threads with those at 8 threads. The gray bars are thought to give a rough idea of the applications' general scaling behavior when going from 2 to 8 threads (i.e., $\frac{T_{2\ threads}}{T_{8\ threads}}$) for the uninstrumented application. Kernels `c_fft`, `c_qsort`, and the loop kernels implement weak scaling, so the expectation is that their execution times are relatively constant. The Jacobi kernels, `c_lu`, `c_mandel`, `c_md`, `c_pi`, and the C++ quicksort kernels (`cpp_qsompX`) use strong scaling, and so the ideal speedup is 4. These values are reported as a reference point for the runtime overhead of our Archer tool. We see a remarkable increase in runtime overhead when going from 2 to 8 threads for some of the strong scaling applications. Especially for the Jacobi kernels, `c_md` and `c_pi`, the overhead increases by a factor 4 or higher. As

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000000

int a[N], b[N];
long int i;

int main(int argc, char **argv)
{
#pragma omp parallel for
  for (i = 0; i < N; ++i) {
    a[i] = i;
  }

  b[0] = 0;

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

Figure 10: OpenMP example containing a data race

expected, the runtime overhead for the Mandelbrot kernel is very low, and for longer runtimes is not discernible since there is almost no access to shared memory in the loop of this kernel.

Note that we omit some measurements from the figure. The `c_jacobi3` kernel has a data race in the break condition and this race is reported by the tool. However, due to this race, the execution time of this kernel is quite random and varies by a factor of 1000; thus we do not report these results. Similarly, the C++ kernels `cpp_qsomp3` and `cpp_qsomp4` rely on a thread-safe implementation of the standard template library (STL). The unsynchronized access in `cpp_qsomp3` leads to a segmentation fault. Thus we report no runtimes for this kernel. Finally, for the `cpp_qsomp4` kernel too many data races with varying stack traces are reported. These let the ThreadSanitizer break with "`Unexpected mmap in InternalAllocator!`". Therefore, no overhead measurement is reported.

The kernel `cpp_qsomp7` provides a task-based implementation of quicksort. We adopted the source code of this kernel to meet the current version of OpenMP task pragma specification, as the original code used an outdated Intel-specific implementation of work-sharing queues. Our experiments with this task example show that we were able to highlight the synchronization points for tasks as well. Data races between tasks that are scheduled on the same thread will be omitted.

In order to present our new fine-grained ThreadSanitizer blacklisting functionality we use the example in Figure 10. The program shows a case with two OpenMP constructs: the first OpenMP *parallel for* is race free since each thread always accesses a distinct location of the array *a*; the second OpenMP construct has a loop-carried data dependency that could generate a data race. In this case, a programmer can blacklist the first OpenMP construct and check only the second one at runtime. Even though this is a simple example, if we set the size of the ar-

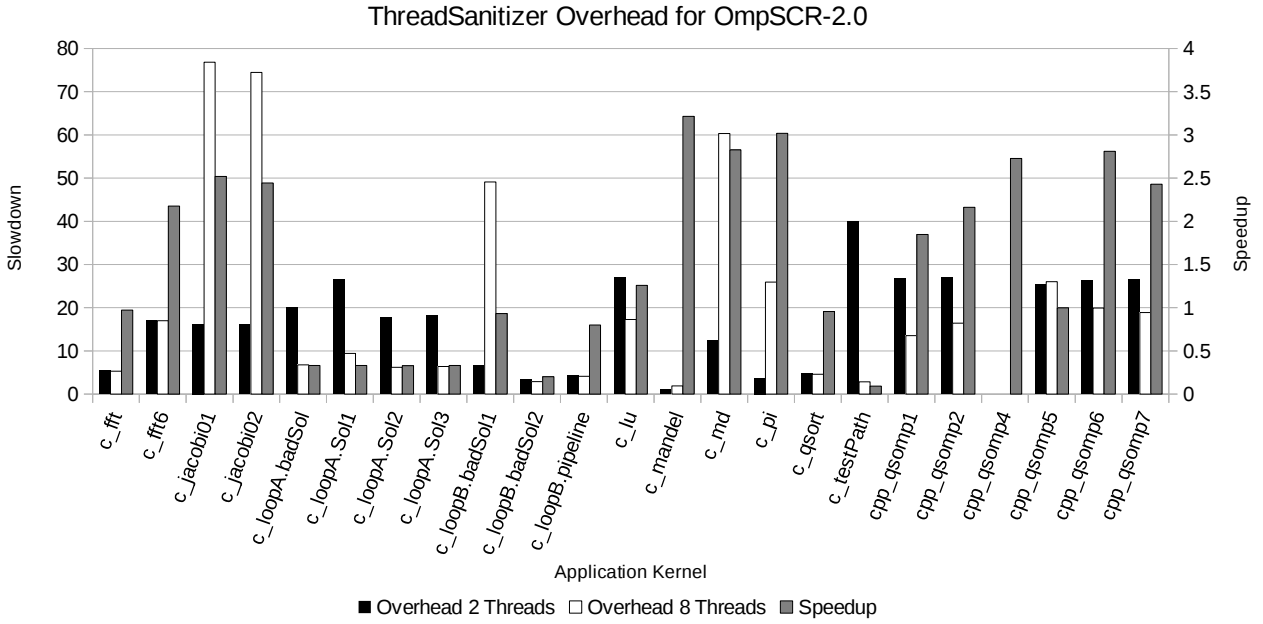


Figure 9: Runtime overhead for OmpSCR collection with 2 threads (left) and 8 threads (middle), up to 77x. The right bars give the speedup for the uninstrumented reference measurement when going from 2 to 8 threads.

	Runtime (s)	**Memory (GB)**
Release	3.373	7.44329
Full blacklisting	13.267	7.53186
Full instrumentation	33.074	37.3479
Blacklist 1st OMP construct	23.674	22.4468

Table 1: Performance results of our line-based blacklist

```
WARNING: ThreadSanitizer: data race (pid=23742)
  Read of size 4 at 0x7f5e0d736924 by thread T21:
    #0 .omp_microtask.1 target.c:19
       (target+0x00000008835f)
    #1 __kmp_invoke_microtask <null>:0
       (libiomp5_tsan.so+0x0000000b9cd2)

  Previous write of size 4 at 0x7f5e0d736924 by
      thread T22:

  Location is global 'b' of size 4000000000 at
      0x7f5d3cd5a620 (target+0x0001c075e924)

  ...

  SUMMARY: ThreadSanitizer: data race target.c:19
      .omp_microtask.1
==================
ThreadSanitizer: reported 2 warnings
```

Figure 11: Report after applying the line-based blacklist

rays *a* and *b* to a very large number (one billion in our case), we can see a big difference in terms of runtime and memory overhead when we run the program in both release mode and under ThreadSanitizer analysis. Blacklisting specific parts of the program allows programmer to obtain better performance since analyzing less code reduces runtime and memory overhead. Note that the original blacklist feature of ThreadSanitizer would not allow blacklisting individual OpenMP constructs. Our extension, in contrast, allows a programmer to specify precisely which lines of code she wants to exclude from the analysis. By blacklisting the first OpenMP construct, we are able to reduce the amount of time and memory that ThreadSanitizer requires to perform the analysis, while still being able to discover the data race in question (see Figure 11).

In Table 1, we list the performance results of the ThreadSanitizer analysis with and without blacklisting. The table compares runtime and memory consumption of the program running in release mode (no analysis performed by ThreadSanitizer) with different levels of instrumentation by ThreadSanitizer. We ran the program on a machine with two processors Intel(R) Xeon(R) CPU E5645 @ 2.40GHz (6 cores, 24 threads total) with 48 GB of RAM. We note that release and full blacklisting (i.e., nothing gets analyzed) are very close in terms of runtime and memory overhead since ThreadSanitizer performs no instrumentation in the full blacklisting case. The difference in runtime and memory overhead between release and full blacklisting is a one-time overhead

that comes from initialization that happens despite full blacklisting of the source code. On the other end of the spectrum, the full instrumentation case shows that ThreadSanitizer runtime and memory overhead is 10x and 6x higher than the release version, respectively. This reflects the performance results published by the ThreadSanitizer authors [16]. The last row of the table shows the case where we apply our fine-grained blacklist instrumentation to the first OpenMP constructs—since there is no data dependency we can blacklist it and run the analysis only for the second OpenMP construct. The numbers show a significant reduction, in terms of runtime and memory overhead, of about 30% less than the fully-instrumented case.

6. LESSONS LEARNED

Our initial approach was to use the OpenMP Tools Interface (OMPT) [6] to insert the annotations. The OMPT interface provides callback functions that are triggered when OpenMP pragmas are executed. This would make the approach applicable to the instrumentation of both the GNU OpenMP library and the Intel OpenMP library. However, we learned that this approach is not compatible with the ThreadSanitizer annotation interface.

As an example, for critical sections, locks and barriers an integer identifier is provided. The properties of this identifier are not specified. It might be the address of the used lock, a consecutively numbered identifier or a random number. On the other hand, as discussed in Section 4.1, ThreadSanitizer uses the shadow memory to propagate the clock values at synchronization points. For the annotation call a common address is needed. Using a hash table and allocating some memory space as unique pointer addresses could solve this issue.

However, there is another critical issue for this approach. The OMPT interface doesn't provide callbacks for all synchronization points. Examples for missing callbacks are the end of critical or locked sections. The callbacks `ompt_event_release_critical` and `ompt_event_release_lock` are specified to be called after the synchronization point. For a valid happen-before mark-up, we need to start the arc just before the lock is released. We agree that with respect to runtime overhead considerations, these callbacks should stay outside the critical regions. Furthermore, some important synchronization points are marked as optional, as they are considered unimportant for performance analysis. Examples of these optional points are the begin and end of a task or even a barrier. In consequence we decided to integrate the highlighting annotations directly into the Intel OpenMP Runtime library.

7. CONCLUSION

In this paper, we successfully combined static and dynamic analysis for race checking between threads and provided a first prototype tool, Archer, targeting OpenMP programs. Our tool is built on top of the well-recognized tool ThreadSanitizer and combines it with an instrumented version of the widely used Intel OpenMP runtime. Our results show that the data race reports generated with Archer are more accurate than most other tools.

Further, we extended the ThreadSanitizer blacklisting feature to allow a finer-grained selection of the code. With this extension, ThreadSanitizer is able to ignore specific lines of code or target specific code regions, rather than entire functions. This feature allows us to maintain the precision of ThreadSanitizer analysis while reducing runtime and memory overhead, creating a novel, highly precise and low-overhead race detection tool for HPC.

Future effort will go into reducing memory overhead and runtime overhead for OpenMP applications. Also, we will integrate the specific semantics of MPI communication calls to make the approach applicable to hybrid OpenMP and MPI applications. In particular, we will focus on asynchronous communication calls and race conditions caused by accessing communication buffers while messages are in flight.

8. REFERENCES

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

[2] Openmp: Support for the openmp language. `http://openmp.llvm.org`.

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

[4] BOARD, O. A. R. OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[5] DORTA, A. J., RODRÍGUEZ, C., DE SANDE, F., AND GONZÁLEZ-ESCRIBANO, A. The openmp source code repository. In *PDP* (2005), IEEE Computer Society, pp. 244–250.

[6] EICHENBERGER, A. E., MELLOR-CRUMMEY, J., SCHULZ, M., WONG, M., COPTY, N., DIETRICH, R., LIU, X., LOH, E., AND LORENZ, D. OMPT: An OpenMP tools application programming interface for performance analysis. In *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. MÃijller, Eds., no. 8122 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2013, pp. 171–185.

[7] GROSSER, T., GROESSLINGER, A., AND LENGAUER, C. Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters 22*, 04 (2012).

[8] JANNESARI, A., BAO, K., PANKRATIUS, V., AND TICHY, W. F. Helgrind+: An efficient dynamic race detector. In *Proceedings of the 23rd international Parallel & Distributed Processing Symposium (IPDPS'09)* (Rome, Italy, 2009), IEEE.

[9] KARLIN, I., KEASLER, J., AND NEELY, R. Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973, August 2013.

[10] LAMPORT, L. A new solution of dijkstra's concurrent programming problem. *Commun. ACM 17*, 8 (Aug. 1974), 453–455.

[11] LATTNER, C. Llvm and clang: advancing compiler technology. *Proc. of FOSDEM* (2011).

[12] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)* (2004), IEEE, pp. 75–86.

[13] SEREBRYANY, K., AND ISKHODZHANOV, T. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA, 2009), WBIA '09, ACM, pp. 62–71.

[14] SEREBRYANY, K., POTAPENKO, A., ISKHODZHANOV, T., AND VYUKOV, D. Dynamic race detection with llvm compiler. In *Runtime Verification* (2012), Springer, pp. 110–114.

[15] SEREBRYANY, K., AND VYUKOV, D. Sanitizer special case list. `http://clang.llvm.org/docs/SanitizerSpecialCaseList.html`.

[16] SEREBRYANY, K., AND VYUKOV, D. ThreadSanitizer, a data race detector for C/C++ and Go. `https://code.google.com/p/thread-sanitizer/`.