IEICE TRANS. INF. & SYST., VOL.E97–D, NO.10 OCTOBER 2014

PAPER

DRDet: Efficiently Making Data Races Deterministic

Chen CHEN[†,††a)], **Kai LU**[†,††], **Xiaoping WANG**[†,††], *Nonmembers*, **Xu ZHOU**[†,††], *Member*, *and* **Zhendong WU**[†,††], *Nonmember*

SUMMARY Strongly deterministic multithreading provides determinism for multithreaded programs even in the presence of data races. A common way to guarantee determinism for data races is to isolate threads by buffering shared memory accesses. Unfortunately, buffering all shared accesses is prohibitively costly. We propose an approach called DRDet to efficiently make data races deterministic. DRDet leverages the insight that, instead of buffering all shared memory accesses, it is sufficient to only buffer memory accesses involving data races. DRDet uses a sound data-race detector to detect all potential data races. These potential data races, along with all accesses which may access the same set of memory objects, are flagged as data-race-involved accesses. Unsurprisingly, the imprecision of static analyses makes a large fraction of shared accesses to be data-race-involved. DRDet employs two optimizations which aim at reducing the number of accesses to be sent to query alias analysis. We implement DRDet on CoreDet, a state-of-the-art deterministic multithreading system. Our empirical evaluation shows that DRDet reduces the overhead of CoreDet by an average of 1.6X, without weakening determinism and scalability.

key words: *determinism, data-race detection, alias analysis, static analysis*

1. Introduction

For shared-memory multithreaded programs, the execution order of synchronization operations and shared memory accesses varies across different executions, even if the same input is given. Nondeterminism makes multithreaded programs hard to develop, test, verify and debug. Deterministic multithreading systems [1] eliminate nondeterminism by deterministically scheduling the synchronization operations and shared memory accesses.

Unfortunately, it is costly to guarantee the determinism of frequent shared accesses. One of the common approaches is to isolate the execution of threads and then enforce inter-thread communication at the deterministic points [2]–[8]. The isolation is implemented by buffering the shared stores and redirecting loads from global memory to local buffer. The frequent buffering, redirection and committing makes the overhead of deterministic multithreading prohibitively high.

To avoid such inefficiency, weak determinism [9], [10] assumes that programs are data-race-free. According to data-race free assumption, no shared access needs to be buffered. Thus, weak determinism is much more efficient than strong determinism. Data-race-free assumption is based on the insight that data races are rare. However, considering that it is hard for developers to avoid introducing data races, we argue that ignoring data races may lead to harmful results.

In this paper we present DRDet, a hybrid approach which efficiently makes data race deterministic. Instead of enforcing deterministic order on all shared memory accesses, we argue that it is sufficient to concern ourselves with data-race-involved accesses. For soundness, data-race-involved accesses should consist of potential data races and any other memory operations which may access the same set of objects. We use a sound static data-race detector to identify all potential data races. The memory objects accessed by these potential data races are called data-race-involved objects. Then, we use alias analysis to detect all shared accesses to these objects.

However, the inherent imprecision of static analyses brings two challenges to DRDet. First, a large fraction of shared accesses are identified as potential data races due to high false positive rates of the static data-race detector. Even worse, too many shared objects are flagged as data-race-involved due to the imprecision of alias analysis. In benchmarks we evaluated, almost all shared accesses are identified as data-race-involved.

DRDet employs two approaches to prune false data-race-involved accesses. Both approaches attempt to minimize the number of accesses to be sent to query alias analysis. First, we adopt *thread specialization* [11] to prune false positives reported by static data-race detector. Having fewer data race warnings leads to fewer data-race-involved objects. Thread specialization specializes a program according to a statically fixed thread count. It is based on the observation that in many multithreaded programs, a large fraction of accesses to shared memory are usually indexed by thread IDs. Static data-race detector usually overestimates the range of shared accesses due to the inability of distinguishing thread IDs by symbolic values. If the thread count is given, thread specialization may statically assign the concrete value to each thread ID. Thus, accesses indexed by thread IDs can be distinguished much easier.

Second, we propose a new buffering strategy called *loads-in-quantum* (LIQ) to reduce the number of accesses need to be buffered. A quantum is the execution phase be-

Manuscript received February 27, 2014.
Manuscript revised May 24, 2014.

[†]The authors are with National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, PR China.
[††]The authors are with the School of Computer, National University of Defense Technology, Changsha, PR China.
a) E-mail: chenchen2011@nudt.edu.cn

DOI: 10.1587/transinf.2014EDP7067

tween two consecutive commit points. Existing buffering strategies [2]–[8] buffer all shared memory accesses. Unlike previous work, LIQ only buffers the stores of data races, as well as loads which (1) happen after the data-race stores and (2) access the same objects in the same quantum. Although LIQ reduces the number of buffered accesses, it results in a relaxed memory consistency model. We discuss if LIQ violates the correctness and determinism of programs.

Same as existing buffering strategies, LIQ is implemented by compile-time instrumentation. We design a compile-time instrumentation algorithm based on LIQ. Although static analyses which we use are imprecise, our algorithm effectively reduces the number of instrumentation points.

Our primary contributions are as follows. First, we propose a new approach to reduce the overhead of deterministic multithreading. Our approach is based on sound static data-race detector and alias analysis. Second, we implemented DRDet on CoreDet [2] and evaluated the effectiveness of DRDet using the SPLASH2 [12] benchmark suits. Our evaluation shows that DRDet improves the performance of CoreDet by 1.6x on average. We find that our two optimizations significantly reduce the number of dynamic instrumentation points.

The remainder of the paper is organized as follows. Section 2 reviews strong determinism and static data-race detection. Section 3 describes the approach we use to prune false data races. In Sect. 4, we describe the loads-in-quantum buffering strategy in detail. Section 5 describes the implementation of DRDet and our experimental results. Section 6 surveys related work and Sect. 7 briefly summarizes the conclusions.

2. Background

In this section we summarize some topics which are relevant to DRDet, including strong determinism and static data-race detection.

2.1 Strong Determinism

Strongly deterministic multithreading not only guarantees the deterministic order for synchronization, but also provides determinism for shared memory access interleavings. A common approach to provide strong determinism is isolating the execution of threads and committing shared memory modifications at deterministic points.

Store buffering is a popular mechanism for isolating threads. As illustrated in Fig. 1, store buffering divides execution into bulk-synchronous quanta. During an individual quantum, stores to shared memory are buffered, and loads to shard memory are redirected to thread-local buffer. Buffered data are committed deterministically at the end of quanta.

Contrary to weak determinism, the overhead of strong determinism is prohibitively high. One of the reasons is that all memory accesses are buffered. CoreDet applies escape analysis to remove instrumentation on accesses which are

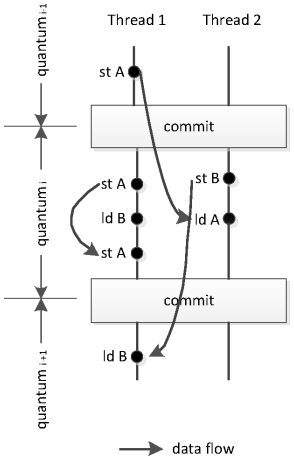

Fig. 1 Threads isolation through store buffering.

provably thread-local. According to the evaluation by Das et al. [13], escape analysis reduces 19.8% dynamic number of instrumentation on average. However, even with escape analysis, the overhead of CoreDet is still roughly 1.2x-6x.

2.2 Static Data-Race Detection

DRDet uses RELAY [14] static data-race detector, which can scales to millions of lines of code. RELAY is a lockset-based data-race detector. A lockset is a set of locks held by program at a program point. A lockset analysis statically computes the lockset of each program point. If two accesses from different threads to the same shared object (1) may happen concurrently, (2) at least one access is a write, and (3) the intersection of the locksets at each point is empty, then a data-race warning is generated.

RELAY uses Steensgaard [15] and Andersen [16] flow-insensitive, context-insensitive, and field-insensitive points-to analyses, which are scalable to large programs. However, these analyses are very conservative.

3. Pruning False Data Races

In this section, we briefly analyze the false positives produced by RELAY, then we summarize *thread specialization* which specializes a program for improving the precision of static data-race detection. We refer to [11] for the details.

3.1 False Positives

As mentioned in Sect. 2, the employment of conservative points-to analyses is one of the main sources of false positives. Considering the tremendous research efforts on points-to analysis over the years, in this paper we mainly focus on other sources except points-to analysis.

Besides the imprecision of points-to analysis, there are two other main sources of false positives. First, arrays are treated as aggregates. RELAY claim that all accesses to the same array are accesses to the same memory object, even

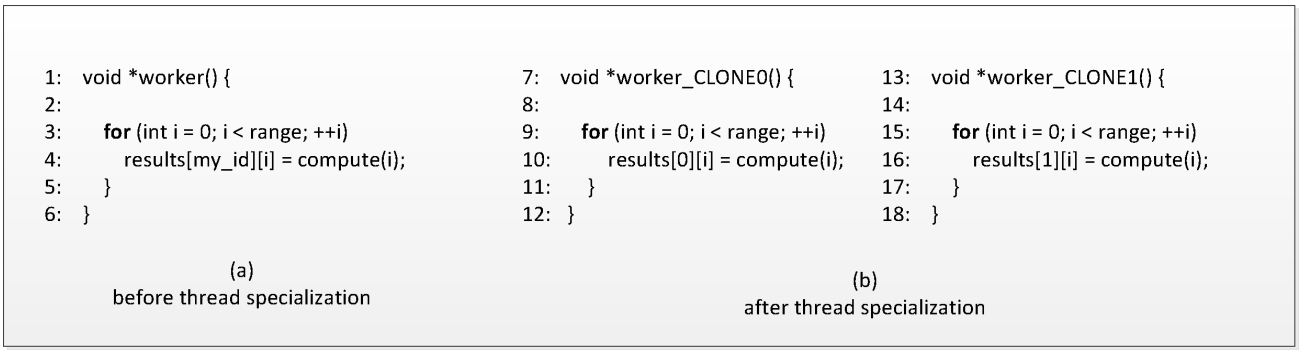

Fig. 2 Thread specialization specializes a program with thread count two.

they actually access different elements. Figure 2 (a) shows an example. RELAY reports data races on *results* in line 4. Actually, the program divides the array *results* into multiple portions and assigns different portion to each worker thread. Thus, different threads would not access the same element in the array *results*. Second, RELAY does not account for the happens-before relationships due to folk-join and barriers. For example, multi-threaded programs usually do initialization before forking worker threads. Shared accesses during initialization would never conflict with any other accesses.

3.2 Thread Specialization

Thread specialization leverages the insight that in most multithreaded programs threads access data according to the ID they are assigned. If we can statically fix thread IDs, then it would be easier to distinguish the objects accessed by each thread. In order to fix thread IDs, we have to fix the thread count at first, since in most programs thread IDs range from zero to the value of thread count. Given a thread count, thread specialization transforms the program into a simplified version. The simplified version has simpler control and data flows, which drastically improves the precision of static data-race detection. Note that the simplified program is equivalent to the original one once the given thread count is enforced. Since the analysis depends on the given thread count, we have to re-analyze a program once a new thread count is given. We argue that it is reasonable based on following insights: (1) the number of cores in the state-of-the-art commodity hardware is small (less than sixty four, usually) and (2) all benchmarks we evaluated achieve peak performance when the thread count is less than or equal to core count.

The process of thread specialization includes two steps. It first specializes the control flow by cloning functions which are reachable by worker threads. Every thread gets its own clone of the reachable functions, so that every statement can only be executed by one single thread. Thus, threads are statically distinguishable. Second, it specializes the data flow by fixing the ID assigned to each thread and replacing the ID variable with the fixed value. Step one makes it impossible that an access races with itself, since every access is only executed by one particular thread. Step two makes inferring of accessing ranges more accurately, since the ID is replaced with constant value.

Figure 2 (b) shows the result of thread specialization on the program of Fig. 2 (a). Function *worker* is cloned twice, assuming that the given thread count is two. In *worker_CLONE0*, the variable *my_id*, which is the index of array *results*, is replaced with constant 0. Similar to *worker_CLONE0*, *my_id* in *worker_CLONE1* is replaced with constant 1.

Now, line 4 in Fig. 2 (a) is transformed into two distinct statements in Fig. 2 (b) (line 10 and line 16). Each statement is mapped to only one active thread, so it is impossible for accesses in any statement to race with itself. Besides, since the indices of the first dimension between two statements are different, we can determine that the access ranges of each statement do not overlap. Thus, two accesses to *results* provably do not race with each other. As a result, the false positives on *results* are pruned.

4. Loads-in-Quantum Buffering Strategy

According to traditional buffering strategies, all shared memory accesses should be buffered. We narrow it down to buffering data-race-involved accesses, which consist of data races and any other accesses to the same set of objects. However, due to the inherent imprecision of alias analysis, the amount of alias accesses remains large, even though we prune a large fraction of false positives via thread specialization. In order to leverage thread specialization we have to design a new buffering strategy.

In this section, we first give the basic correctness and deterministic criteria which a buffering strategy should obey. Then we propose our buffering strategy and discuss why it does not violate the correctness and deterministic criteria. Finally, we describe our compile-time instrumentation algorithm, which implements the buffering strategy.

4.1 Correctness Criterion

For accesses to the same memory object, if a store S is buffered, then all loads between S and the next store must be buffered, namely these loads must read value from buffer. Similarly, if S directly writes value to shared memory, then

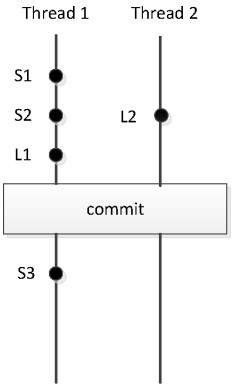

Fig. 3 An example of buffering strategy.

all loads between S and the next store must read value from shared memory. Considering that in buffering based deterministic multithreading all buffered values are committed at quantum boundaries, we define the basic correctness criterion as follows.

Correctness criterion: In a quantum, if a store is buffered, then all the accesses which happen after the store and access the same object in the same thread must be buffered.

The basic correctness criterion is sufficient to guarantee correctness in strong determinism. Considering accesses in the same thread, if a store is buffered, the accesses (either stores or loads) which happen after the store must be buffered for consistency. Accesses in other quanta do not need to be buffered since buffered data are committed to shared memory at the boundary of quantum. It is correct that the accesses which happen before the buffered store directly access the global memory, as long as the address has not been buffered yet. Correctness of programs is independent of inter-thread buffering decision, since threads are isolated. Taking Fig. 3 as an example, if store S2 is buffered, then load L1 must be buffered while store S1 does not have to. Whether or not to buffer store S3 and load L2 does not depend on the status of store S1 and S2, since S3 is in other quantum and L2 is in other thread.

4.2 Deterministic Criterion

Traditional buffering strategies believe that all data races should be buffered. However, we argue that, for a data race, buffering one of the stores is sufficient to make the data race deterministic. If two accesses of the data race reside in different quanta, the global synchronization between two quanta makes the data race deterministic, thus no buffering is needed. If they reside in the same quantum, buffering one store is sufficient to introduce a deterministic happens-before relationship to the data race. The other access does not need to be buffered, since it always happens before the buffered store. For example, load L2 in Fig. 3 races with store S1 and store S3. S1 and L2 are in the same quantum. Buffering S1 results in that L2 always happens before S1. S3 and L2 are in different quanta, thus the race between S3 and L2 is eliminated by the global barrier between quanta. We summarize the basic deterministic criterion as below.

Deterministic criterion: In a quantum, at least one of the stores in a data race must be buffered.

4.3 Buffering Strategy

We design the *loads-in-quantum* (LIQ) buffering strategy to reduce the amount of buffered accesses. For each data race, LIQ operates in two steps. Step I, LIQ buffers store(s). Step II, LIQ buffers all loads which happen after the stores and access the same object in the same quantum.

LIQ does not violate correctness and deterministic criteria. First, according to deterministic criterion, buffering data-race store(s) makes data race deterministic. Second, according to correctness criterion, we should buffer following stores and loads. In fact, stores which happen after the data-race store access the same object as the data-race store does, so they are data races too. Thus, these stores will be buffered in other rounds anyway. It is sufficient to only buffer loads which happen after the data-race store in step II. Taking Fig. 3 as an example, stores S1, S2 and S3 access the same object. S1 and S2 reside in the same quantum. S3 is in the next quantum. We do not take into account S2 in step II because S2 is a data-race store and will be buffered in other round. S3 resides in other quantum, thus we do not have to take into account S3 in current round.

4.4 Compiler-Time Instrumentation

We implement a compile-time instrumentation algorithm based on LIQ. Although the idea of LIQ is simple, it is challenging to deal with the imprecision of static analyses which we employ. The sources of imprecision are (1) potential data races, (2) may-alias analysis, and (3) conservatively inferred quantum boundaries. During compile time, the algorithm iterates over all potential data-race stores. Before detecting loads dominated by these potential data-race stores in the same quantum, DRDet has to infer the quantum boundaries at first. For each potential data-race store, DRDet conservatively assumes that the quantum starts from the location of the store. In this quantum, DRDet enumerates all loads which are dominated by the store and may access the same object as the store does. DRDet performs a breadth-first traversal of basic blocks in the inter-procedural CFG of the quantum. The traversal starts with a budget of full quantum size and computes the remaining budget according to the amount of work in each basic block. The traversal ends when the budget is exhausted or encountering an external call.

The algorithm is presented in Fig. 4. The input is a set of all potential data-race stores, and the output is a set of shared loads which dominated by potential data-race stores. States are named by a pair of (bb, budget), where bb represents the current basic block, and budget represents the amount of remaining instructions in this quantum. Every time a new basic block is traversed, the remaining budget

```
EnumDominatedLoads (PDRS: potential data-race stores) {
 dominatedLoads = {}
 foreach (s in PDRS) {
  initState.bb = s.parentBB()
  initState.budget = QuautumSize
  worklist = {initState}
  while (!worklist.empty()) {
   curState = worklist.pop()
   curBB = curState.bb
   //if current basic block has been visited with a
   //bigger budget, no new basic block will be
   //introduced, so prune current state
   if (curBB visited &&
       curState.budget <= curBB.MaxBudget)
    continue
   else {
    curBB.MaxBudget = curState.budget
    curState.budget -= curBB.work
    worklist.add(SuccessorBB(curBB))
    if (curBB has not yet been visited) {
     foreach (shared load l in curBB) {
      if (mayAlias(l,s))
       dominatedLoads.add(l)
     }
    }
   }
  }
 }
 return dominatedLoads
}
```

Fig. 4 Enumerating dominated loads for every potential data-race stores.

```
SuccessorBB (curState) {
 newStates = {}
 curBB = curState.bb
 //if curBB has no successor, first consult call stack
 //then the call graph
 if (curBB has no successor) then
  if (!curState.callStack.empty()) then
   newState.bb =  curState.callStack.pop()
   newState.budget = curState.budget
   newStates.add(newState)
  else { //call stack is empty, iterate all callees
   foreach(c in callees of curBB.getParent()) {
    newState.bb = c.entry()
    newState.budget = curState.budget
    newState.add(newState)
   }
  }
 else { //curBB has successors, iterate the successors
  foreach (succ in successors of curState.bb) {
   newState.bb = succ
   newState.budget = curState.budget
   newState.add(newState)
 }
 return newStates
}
```

Fig. 5 Iterating successors of a basic block.

is updated and the shared loads within the basic block are checked by alias analysis to see if they may assess the same object. For basic blocks which have been visited before, if the remaining budget of current state is smaller than or equal to the max budget this basic block has ever been visited with, which means current state would not introduce any new basic block, then current state is pruned. Otherwise, successors of current basic block is computed and inserted to the work list.

Figure 5 shows the algorithm for iterating successors of a basic block. To deal with inter-procedure analysis, DRDet maintains a call stack which indicates where to return. When a return instruction is reached, DRDet consults the call stack to get successors. Note that the analysis does not start from the entry but the middle of the program, so the first function does not have the return location in the call stack. In this case, DRDet treats the entries of all potential callees of the first function as successors. Callees are computed by consulting call graph and alias analysis.

Since the static analyses which we employ are sound, our instrumentation algorithm does not introduce any false negative. First, the soundness of the static data-race detector guarantees that all data-race stores are instrumented. Second, the soundness of alias analysis guarantees that DRDet instrument all loads which happen after data-race stores and access the same object.

Although the inherent imprecision of static analyses inevitably introduces false positives, these false positives do not violate the sematic of programs. The reasons are: (1) it is safe to conservatively buffer redundant loads; (2) all potential data races are treated as real data races; and (3) DRDet instruments no store except potential data-race stores.

5. Evaluation

Our main goal in this section is to show that DRDet improve the performance of deterministic execution without weakening determinism and scalability. We implemented DRDet on CoreDet, and compared DRDet with native CoreDet. The performance and scalability data is shown. Furthermore, we demonstrate the effectiveness of thread specialization and loads-in-quantum buffering strategy.

5.1 Methodology

We implemented the idea of DRDet on CoreDet with store buffering configuration. Programs were analyzed by static data-race detector for generating data race reports before being compiled by DRDet. The static data-race detector we used was RELAY with thread specialization support. Thread specialization transformed programs according to the given thread count. RELAY analyzed the transformed programs to generate data race warnings. Based on these warnings, DRDet instrumented data-race-involved accesses during compilation. We reused the runtime of CoreDet to implement a complete strong deterministic multithreading system.

We evaluated DRDet using seven programs from SPLASH-2 [12] benchmark suites: *fft*, *water*, *barnes*, *ocean*, *radix*, *lu* and *fmm*. Scaled inputs were selected to insure that every program runs for at least about a minute with one thread.

We ran our experiments on an AMD server with a 2.2 GHz, 12-core CPU (AMD Opteron 6174) and 16 GB physical memory, running Linux kernel version 2.6.31.5. We used LLVM 3.0 [17] as compiler infrastructure. CoreDet

was ported from LLVM 2.6 to LLVM 3.0 for comparison. We test the determinism of DRDet by running racey deterministic stress test [18] for 1000 times. During the execution of racey, we disabled the ASLR (Address Space Layout Randomization) of Linux. DRDet reports the same results for all 1000 runs.

5.2 Performance and Scalability

We compared the overheads and scalability of DRDet with CoreDet, using DMP-B scheme (store buffering). We enforce the same configuration on DRDet and CoreDet. The values we choose for *granularity* and *quantum size parameters* are optimal on CoreDet.

Figure 6 shows DRDet's performance results when thread specialization and loads-in-quantum are enabled. Each bar shows DRDet normalized to CoreDet with the same number of threads, showing how much overhead is reduced by DRDet. Overall, the speedup over CoreDet for 2,4 and 8 threads range from 1.1x-7.6x, 1.1x-6.4x and 1.1x-5.3x respectively. The harmonic means across all benchmarks for 2, 4, and 8 threads are 1.6x, 1.7x and 1.7x, respectively. The performance of *water* is significantly improved because that DRDet removes almost all access instrumentation.

Figure 7 compares the scalability of DRDet with CoreDet. Each bar is normalized to the same configuration with 2 threads. Overall, DRDet scales as well as DMP-B scheme of CoreDet. Note that among three schemes in CoreDet, DMP-B scales best. We conclude that DRDet improves the performance of DMP-B without sacrificing scalability.

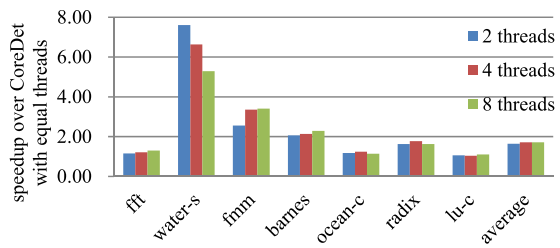

Fig. 6 Overheads relative to CoreDet with the same numbers of threads.

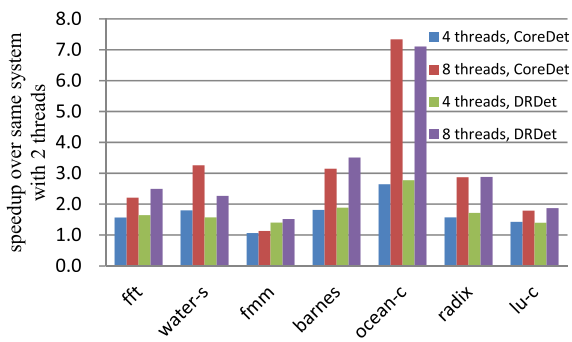

Fig. 7 Scalability.

5.3 Effectiveness of Thread Specialization and LIQ

We analyzed the effectiveness of thread specialization and LIQ on DRDet's performance. Static data-race detectors lack the ability to precisely distinguish threads, which results in a lot of false data races. Thread specialization transforms a multithreaded program into a simplified version according to the given thread count. Analysis on the simplified program generates more precise results (Sect. 3). Loads-in-quantum buffering strategy tries to buffer as fewer accesses as possible. To this end, LIQ only checks loads which are dominated by data-race stores before quantum boundaries (Sect. 4).

We compared three different sets of optimizations. The basic optimization uses RELAY to detect potential data races and then uses alias analysis to detect all accesses to the same set of memory objects that accessed by data races. The opt2 optimization introduces thread specialization to prune false data races, and uses the same buffering strategy as the basic analysis. Unlike opt2, opt3 buffers accesses according to LIQ.

Figure 8 shows the performance overhead of DRDet normalized to CoreDet with different optimizations. All benchmarks were run with 4 threads. For all benchmarks we evaluated, the basic optimization contributed quite little to overall overhead reduction. The harmonic mean of speedup with opt2 is 1.3x. Thread specialization reduced potential data races, which led to the reduction of the amount of data-race-involved memory objects. Thus, accesses to the pruned objects were ignored by opt1. When we further employed LIQ buffering strategy in opt2, the average speedup of DRDet is up to 1.6x.

Applications such as *water*, *fmm* and *barnes* benefit significantly from thread specialization. In these applications, most of reported data races are accesses on array elements indexed by thread IDs. These false data races are soundly pruned by thread specialization. In contrast, thread specialization is ineffective on applications such as *fft*, *lu* and *ocean*. In these applications, many accesses on arrays are indexed by thread IDs indirectly. For example, some indices are functions of thread ID. Thread specialization could not account for such indices.

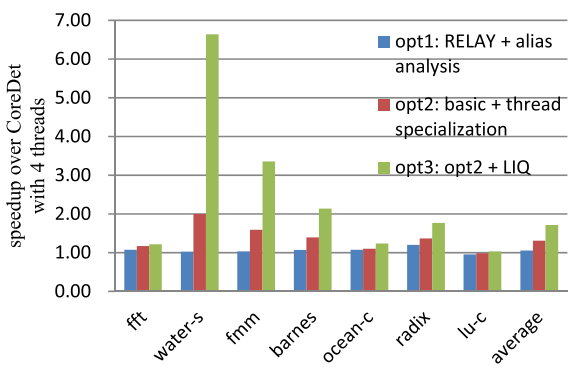

Fig. 8 Speedup over CoreDet with different sets of optimizations.

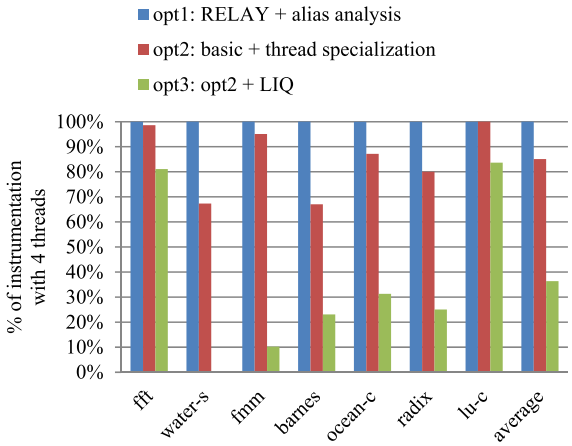

Fig. 9 Proportion of instrumentation points with different sets of optimizations.

We observe that the effectiveness of LIQ depends on the results of thread specialization. For example, opt2 results in a speedup of about 2x on water, while opt3 provides a 7x speedup. In contrast, for *fft*, opt2 gains a speedup of about 1.1x, while opt3 only achieves a 1.2x speedup. Since LIQ instruments loads which happen after each data-race store, fewer data-race stores leads to fewer instrumented loads.

Figure 9 shows the proportion of dynamic number of instrumented accesses with respect to the total number of dynamic shared memory operations. In general, the proportions in Fig. 9 for different optimizations are consistent with the results in Fig. 8. REALY flags a large fraction of shared memory operations as potential data races. Alias analysis based on these potential data races identifies almost all shared accesses as data-race-involved. On average, opt2 reduces 15% of dynamic number of instrumentation, which indicates the benefit of thread specialization. By further employ the LIQ buffering strategy, DRDet reduces the proportion of instrumentation with respect to total memory accesses down to 36% on average.

Note that for some programs, the reduction of dynamic instrumentation does not yield a proportional performance benefit. For example, opt3 reduces 75% of dynamic instrumentation points on *radix*, but only improves the performance by 1.45x. Similarly, opt3 reduces 69% of instrumentation on *ocean*. However, the performance gain is only about 1.07x. The main reason of the disproportionateness is that, instrumentation reduction may introduce quantum imbalance between threads. Recall that CoreDet enforces global barriers at the end of each quantum, it is essential to keep load balance between threads. DRDet introduces load imbalance on some programs because it reduces different amount of instrumentation for different threads. However, results in Fig. 9 are consistent with the overhead results in Fig. 8 in general. This indicates that the benefits of buffering reduction far outweigh the quantum imbalance it introduces.

Table 1 Analysis time. LOC shows the lines of code in each application. We show the execution time of program analysis with thread specialization (SPEC) and the execution time of original RELAY (RELAY). The analysis time is measured in seconds.

application	LOC	SPEC	RELAY
fft	0.9K	8.45	8.074
water-s	1.9K	10.306	11.681
fmm	3.2K	12.329	12.58
barnes	2.0K	9.683	10.349
ocean-c	4.0K	21.399	13.02
radix	0.9K	8.568	8.059
lu-c	0.9K	8.546	8.221

5.4 Analysis Time

Table 1 shows the execution time of DRDet's program analysis with comparison to the original RELAY. Overall, like RELAY, the execution time of thread specialization largely depends on the size of program. Our results illustrate that thread specialization incurs negligible overhead, compared to RELAY. Note that in some cases (e.g., *water*, *fmm* and *barnes*) thread specialization does not increase but reduce the analysis time. The key reason is that thread specialization drastically prunes false positives, resulting in a significant decrease in the report generation time. Among these applications, the analysis time for *ocean* is the largest because (1) the LOC of *ocean* is the largest and (2) thread specialization is relatively ineffective on *ocean*.

6. Related Work

Deterministic Multithreading. Deterministic multithreading provides two levels of determinism: strong determinism and weak determinism [9], [10]. Weak determinism avoids heavy-weight tracking of memory accesses. However, multithreaded programs are hard to guarantee data-race-free. There is a large body of work that provides strong determinism [1]–[7], [19], [20], [31], which is commonly implemented by isolating threads and enforcing thread communication at deterministic points. Store buffering [2]–[8] is a common mechanism to provide thread isolation. Traditional buffering strategies usually buffer all shared memory accesses. CoreDet employs an escape analysis to prune unescapable accesses. DThreads [21] employs a different mechanism, which treats threads as processes and leverages OS memory protection to enforce isolation between threads. Accesses to shared memory are redirected to local address spaces, and are merged to global memory at synchronization points. For many programs, DThreads is efficient due to avoidance of instrumentation and reduction of false sharing. However, since synchronization operations are treated as barriers, DThreads is likely to introduce load imbalance for programs with irregular synchronization patterns. DRDet isolates threads through store buffering. Unlike previous work, DRDet's buffering strategy employs a number of static analyses to reduce the number of instrumentation points. Although the employment of static analyses increases analysis time, we believe it is a worthy tradeoff

for better runtime performance.

There has been previous work to reduce instrumentation overhead of deterministic multithreading. CoreDet employs an escape analysis to remove instrumentation on accesses to thread-local data. Das et al. [13] proposes three techniques to reduce logging overhead for deterministic execution. They prunes three categories of accesses from instrumentation, including (1) accesses executed when only a single thread is running, (2) accesses to multi-threaded read-only memory and (3) accesses in a loop to arrays whose address does not change. Unlike [13], DRDet directly targets at data races. Besides, DRDet does not require any developer-provided hint.

Stable Multithreading. Stable multithreading [22]–[24] makes data races deterministic by reusing the recorded thread interleavings for the same input. Peregrine [23] enforces hybrid schedules at runtime for deterministic. A hybrid schedule consists of mem-schedule, the happens-before order of memory operations, and sync-schedule, the happens-before order of synchronization operations. Mem-schedule is enforced when a code region is likely to contain data races, while sync-schedule is enforced otherwise. Mem-schedule is computed by schedule specialization [25] based on the sync-schedule, which facilitates the detection of data races by simplifying control and data flows. Stable multithreading is efficient due to enforcing of hybrid schedules. However, the recorded schedules do not guarantee to cover all inputs. If an input gets no mapping schedule in the schedule database, the execution is nondeterministic. DRDet guarantees determinism for thread counts which have been analyzed by thread specialization. The set of thread count is much more small than the set of schedules.

Static Data-Race Detection. A plethora of work has been devoted to creating precise and scalable static data-race detectors [14], [26]–[29]. Many of them try to explore the tradeoff between precision and scalability. RELAY is sound and scalable to millions of lines of code, which is suitable for DRDet to detect all potential data races in large programs.

False Positive Pruning of Static Analysis. RELAY provides several simple filters to prune false positives. However, all these filters are unsound. Chimera [30] employs RELAY to detect potential data races and instruments them with weak-lock to make program data-race-free. To reduce the number of weak-locks, Chimera merges consecutive weak-locks by enlarge the region weak-locks protect. Similar to Chimera, DRDet uses RELAY. However, the optimizations in Chimera cannot be ported to DRDet, since Chimera does not directly prune false positives. Schedule specialization [25] specializes a program toward a small set of recorded thread interleavings. Static analysis on specialized program produces much more precise results. However, schedule specialization requires recording a set of schedules in advance. Contrary to schedule specialization, thread specialization does not need any information about schedule. Besides, the set of thread count is much small than the set of schedules, since every thread count maps at least one distinct schedule.

7. Conclusion

In this paper, we propose a novel approach to reduce the overhead of deterministic multithreading. Our approach leverages a static data-race detector to detect potential data races, and uses alias analysis to detect all accesses to the data-race-involved memory objects. Instrumentation overhead is drastically reduced by two critical optimizations.

Although DRDet focuses on reducing instrumentation overhead of deterministic multithreading systems, the idea of DRDet can be easily ported to deterministic replay or dynamic data-race detection.

Acknowledgments

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61170261, 61103193 and 61402492.

References

[1] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared memory multiprocessing," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, USA, pp.85–96, 2009.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," Proc. Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, Pittsburgh, Pennsylvania, USA, pp.53–64, 2010.

[3] T. Bergan, N. Hunt, L. Ceze, and S.D. Gribble, "Deterministic process groups in dOS," Proc. 9th USENIX Conference on Operating Systems Design and Implementation, pp.177–192, 2010.

[4] A. Amittai, W. Shu-Chun, H. Sen, and F. Bryan, "Efficient system-enforced deterministic parallelism," Proc. 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, BC, Canada, pp.1–16, 2010.

[5] D.R. Hower, P. Dudnik, M.D. Hill, and D.A. Wood, "Calvin: Deterministic or not? Free will to choose," High Performance Computer Architecture (HPCA), pp.333–344, 2011.

[6] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, "RCDC: A relaxed consistency deterministic computer," Proc. Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, California, USA, pp.67–78, 2011.

[7] X. Zhou, K. Lu, X. Wang, and X. Li, "Exploiting parallelism in deterministic shared memory multiprocessing," J. Parallel Distrib. Comput., vol.72, no.5, pp.716–727, 2012.

[8] N. Hunt, T. Bergan, L. Ceze, and S.D. Gribble, "DDOS: Taming nondeterminism in distributed systems," Proc. Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp.499–508, 2013.

[9] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.97–108, 2009.

[10] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G.A.

Gibson, and R.E. Bryant, "Parrot: A practical runtime for deterministic, stable, and reliable threads," Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles, pp.388–405, 2013.
[11] C. Chen, K. Lu, X. Wang, X. Zhou, and L. Fang, "Pruning false positives of static data-race detection via thread specialization," Advanced Parallel Processing Technologies, pp.77–90, Springer, 2013.
[12] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," ISCA, pp.24–36, 1995.
[13] M. Das, G. Southern, and J. Renau, "Reducing logging overhead for deterministic execution," in Workshop on Determinism and Correctness in Parallel Programming (WoDet), 2013.
[14] J.W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.205–214, 2007.
[15] B. Steensgaard, "Points-to analysis in almost linear time," Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.32–41, 1996.
[16] L.O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. thesis, DIKU, University of Cophenhagen, 1994.
[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," CGO, pp.75–86, 2004.
[18] M. Hill and M. Xu, "Racey: A stress test for deterministic execution," Available: http://www.cs.wisc.edu/~markhill/racey.html
[19] E.D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++," OOPSLA, pp.81–96, 2009.
[20] K. Lu, X. Zhou, X. Wang, W. Zhang, and G. Li, "RaceFree: An efficient multi-threading model for determinism," PPoPP, pp.297–298, Shenzhen, China, 2013.
[21] T. Liu, C. Curtsinger, and E.D. Berger, "DTHREADS: Efficient deterministic multithreading," Proc. 22nd ACM Symposium on Operating Systems Principles, pp.327–336, 2011.
[22] H. Cui, J. Wu, and J. Yang, "Stable deterministic multithreading through schedule memoization," 9th OSDI, pp.207–222, 2010.
[23] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, "Efficient deterministic multithreading through schedule relaxation," SOSP '11, Cascais, Portugal, pp.337–351, 2011.
[24] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu, "Determinism is not enough: Making parallel programs reliable with stable multithreading," Commun. ACM, vol.57, no.3, pp.58–69, 2014.
[25] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang, "Sound and precise analysis of parallel programs through schedule specialization," Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.205–216, 2012.
[26] D. Englerand and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," 19th ACM Symposiumon Operating Systems Principles (SOSP), pp.237–252, Oct. 2003.
[27] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.13–22, 2009.
[28] P. Pratikakis, J.S. Foster, and M. Hicks, "Locksmith: Practical static race detection for C," ACM Trans. Programming Languages and Systems (TOPLAS), vol.33, pp.1–55, 2011.
[29] N. Sterling, "Warlock: A static data race analysis tool," USENIX Winter Technical Conference, pp.97–106, 1993.
[30] D. Lee, P.M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid program analysis for determinism," PLDI, Beijing, China, pp.463–474, 2012.
[31] X. Zhou, K. Lu, X. Wang, W. Zhang, K. Zhang, X. Li, and G. Li, "Deterministic message passing for distributed parallel computing," IEICE Trans. Inf. & Syst., vol.E96-D, no.5, pp.1068–1077, May 2013.

Chen Chen received the B.S. and M.S. degrees in 2006 and 2010, respectively, from the School of Engineering, Air Force Engineering University, Xi'an, China. He currently pursues the Ph.D. degree in the School of Computer Science, National University of Defense Technology, Changsha, China. His research interests include program analysis and parallel computing.

Kai Lu received the B.S. and Ph.D. degrees in 1995 and 1999, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China. He is now a Professor in the School of Computer Science, National University of Defense Technology. His research interests include operating system, parallel computing and security.

Xiaoping Wang received his B.S., M.S., and Ph.D. degree in the School of Computer Science from National University of Defense Technology, China, in 2003, 2006, and 2010, respectively. He is now an assistant professor in the School of Computer Science at National University of Defense Technology. His research interests include operating system and sensor networking.

Xu Zhou received the B.S., M.S. and Ph.D. degrees in the School of Computer Science, National University of Defense Technology, Changsha, China, in 2007, 2009 and 2013, respectively. He is now an assistant professor in the School of Computer Science at National University of Defense Technology. His research interests include operating system and parallel computing.

Zhendong Wu received the B.S. and M.S. degrees in 2009 and 2011, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China, where he is currently pursuing the Ph.D. degree. His research interests include parallel computing and software reliability.