Accurate and Efficient Filtering for the Intel Thread Checker Race Detector

Paul Sack
Department of Computer Science
University of Illinois at Urbana-Champaign
paulsack@cs.uiuc.edu

Brian E. Bliss, Zhiqiang Ma, and Paul Petersen
Intel Corporation
{brian.e.bliss, zhiqiang.ma, paul.petersen}@intel.com

Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
torrellas@cs.uiuc.edu

ABSTRACT

Debugging data races in parallel applications is a difficult task. Error-causing data races may appear to vanish due to changes in an application's optimization level, thread scheduling, whether or not a debugger is used, and other effects. Further, many race conditions cause incorrect program behavior only in rare scenarios and may lie undetected during software testing.

Tools exist today that do a decent job in finding data races in multi-threaded applications. Some data-race detection tools are very efficient and can detect data races with less than a 2x performance penalty. Most such tools, however, do not provide enough information to the user, require recompilation, or impose other usage restrictions. Other tools, such as the one considered in this paper (Intel's Thread Checker), provide users with plenty of useful information and can be used with any application binary, but have high overheads – often over 200x. It is the goal of this paper to speed up Thread Checker by filtering out the vast majority of memory references that are highly unlikely to be involved in data races. In our work, we develop filters that filter 90-100% of all memory references from the data-race detection algorithm, resulting in speedups of 2.2-5.5x, with an average improvement of 3.3x.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools*

General Terms

Data-race detection

1. INTRODUCTION

Debugging data races is a difficult task. Even when programmers know the symptoms of a data race, it can be difficult to zero-in on the cause. Erroneous behavior may only manifest itself under certain conditions. Changing compiler optimization levels, using a debugger, or incurring slightly different thread scheduling by the OS can change the thread interleaving such that a bug's symptoms vanish. Further, exhaustive software testing can miss some races that can appear under extremely rare circumstances, but can still be important in critical applications.

For these reasons, developers increasingly turn to tools that use data race-detection algorithms to find race conditions in multi-threaded applications. Such tools ease data race debugging, since they can pinpoint two conflicting memory references to the same memory location that are not properly ordered with synchronization.

There are two common types of algorithms used in finding data races in multi-threaded applications. They are lockset-based algorithms, such as Eraser [13], and vector clock-based algorithms, such as RecPlay [12] and Intel's Thread Checker [5]. Some algorithms use a hybrid approach, such as RaceTrack [16] and Choi *et al* [2].

Such tools can uncover many data races that otherwise would lie undetected or would be difficult to debug. Many such tools have reasonable overheads – on the order of 2x slowdowns – but do not provide the user with much useful information or have limited usage models. Other tools, such as the one considered in this paper (Intel's Thread Checker), provide the user with an abundance of useful information for debugging and have few usage constraints, but have high performance costs, as we will see.

In this work, we develop an integrated filter for Thread Checker that can cut the cost of data race-detection without sacrificing detection accuracy. The filter that we develop filters, on average, 98% of the memory references in the SPLASH-2 benchmarks without missing any data races. This improves the performance of Thread Checker by an average of 3.3x, or a maximum of 5.5x.

In section 2, we introduce and compare the two common data-race detecting algorithms. We then focus on Intel's Thread Checker and examine its sources of overhead. In section 3, we present an overview of our filter design. In section 4, we explain how it is implemented. In section 5, we discuss our experimental setup. In section 6 we present our evaluation. In section 7, we discuss related work, and

ASID'06 October 21, 20006, San Jose, California, USA.

we conclude in section 8.

2. MOTIVATION

A conflicting access pair consists of two memory accesses by two different threads to the same address, at least one of which is a store. A data race occurs when there is a conflicting access pair and the two accesses are not ordered using a synchronization operation, such as obtaining a lock or creating a thread.

2.1 Lockset algorithm

The lockset algorithm is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Strictly speaking, it does not detect data races, but detects violations of a *locking discipline*. A locking discipline is a consistent set of locks held while accessing a shared variable. The locking discipline ensures that a shared variable is either always read and written while holding a common lock, or only read. It operates by intersecting the currently-held set of locks (lockset) with the saved lockset. If the intersection is null, then the locking discipline has been violated, and a diagnostic message is reported to the user. Otherwise, the intersected lockset is saved. Eraser [13] is a commonly-used lockset-based data-race detector.

One problem with this algorithm, as described above, is in shared-variable initialization. A common programming idiom is to have a main thread allocate and initialize a data structure and then spawn child threads that access the structure in some manner. The initialization is often done without holding a lock before the child threads are created – thread creation synchronizes the conflicting accesses by the main thread and the child threads. When a child thread accesses the data structure, it is observed that the currently held lockset and the intersection of the lockset during initialization, i.e., the empty set, is zero, and thus the locking discipline has been violated and a diagnostic is output.

Another trivial shortcoming is the case of a shared variable that is initialized by the main thread before child threads are created, and then only read. An example is a variable containing the PID of the application. In this case, the variable is again accessed without holding any locks and a violation is detected – even though it is only read.

To solve this scenario, Eraser employs the state machine illustrated in Figure 1. Variables are either Uninitialized, Private, Shared Read-Only, or Shared Read-Write. Upon the first access to a variable (R1, W1), the variable enters the Private State. From this state, upon further accesses by the same thread (R1, W1), the variable remains in the Private state. Should another thread read the variable (R'), the variable enters the Shared Read-Only state. Should another thread write (W') the variable, the variable enters the Shared Read-Write state. Finally, from the Shared Read-Only state, a write by any thread (W) takes the variable to the Shared Read-Write state, but a read by any thread (R) does not change the state. Lockset analysis is only performed in the two shared states, and locking-discipline violations are only reported in the final Shared Read-Write state. The addition of this state machine reduces the number of false positives Eraser reports.

Unfortunately, the lockset algorithm cannot handle applications that use other forms of synchronization, such as forks and joins, semaphores, or barriers. This is why vector-clock based algorithms, such as RecPlay [12] and Intel's Thread Checker [5], are used for finding races in applications which use other forms of synchronization.

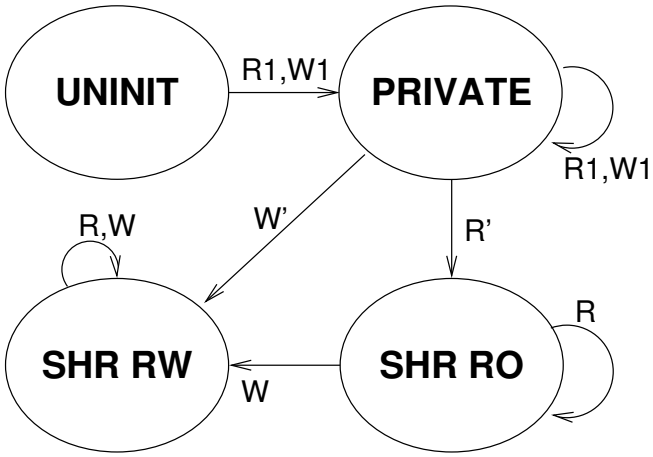

Figure 1: Eraser state machine.

2.2 Vector clock algorithm

Vector clocks are complex in comparison to locksets. It is this complexity that allows them to handle all kinds of synchronization. This also makes vector clock algorithms more difficult to understand.

In vector clock algorithms, each thread executes *segments*. A segment is defined as all the instructions executed by a thread between synchronization operations.

A simple definition of a vector clock is that it is a type of clock that allows one to specify an ordering amongst these segments. The vector clock specifies whether, given two segments a and b running on different threads, a precedes b, b precedes a, or if a and b overlap.

Each thread has a vector clock that it uses to order its segments with respect to the other threads' segments. Further, each variable has two vector clocks, to keep track of the last time each thread read it and wrote to it.

Data race detection using vector clocks is simple. Each time a thread reads from a variable, it checks that the thread's vector clock is ordered with respect to the variable's write vector clock – which records information on the last write to the variable by all threads. Each time a thread writes to a variable, it checks that the thread's vector clock is ordered with respect to the variable's read and write vector clocks – which record information on the last read and write to the variable, respectively, by all threads. If two vector clocks are not ordered, we have a race – the conflicting access pair could have occurred in either order.

The use of vector clocks incurs an enormous space overhead. For a system with n threads, each vector clock must contain n clock fields, and each field might be 4 bytes wide or more. If threads are dynamically created and destroyed, n can grow to be very large.

We refer the reader to the literature [4, 9, 12] for further details on the operation of a vector clock-based data race detector and some variations and optimizations. A good explanation of vector clocks is found in [7]. A detailed understanding of vector clock algorithms is not necessary for following the rest of this paper.

2.3 Overheads of Intel's Thread Checker

Some data race detectors, including Intel's Thread Checker [5], are quite slow. As seen in Figure 2, Thread Checker has an average slowdown of 233x in the SPLASH-2 applications with 4 threads running on a 4-processor machine, and a maximum slowdown of 485x. There are many reasons why Thread Checker is slower than some competing data race detectors. One cause is that Thread Checker provides the call stack for the two racing memory references. Maintaining the call stack can be very time-consuming, but is invaluable in providing programmers with context information to aid in debugging. Secondly, the reported slowdowns are relative to compiled, highly-optimized, well-parallelized C code – not code running in a virtual machine or managed runtime environment. Many race-detection optimizations possible for type-safe managed languages do not work with binary code written in non-type-safe languages, such as C. Third, Thread Checker works with virtually any Linux or Windows binary application with dynamically-loaded and unloaded libraries.

The central Thread Checker vector clock algorithm requires the following information for each reference:

- the address of the reference
- whether it is a load or store
- whether it is a synchronization operation
- the line of code
- the call stack

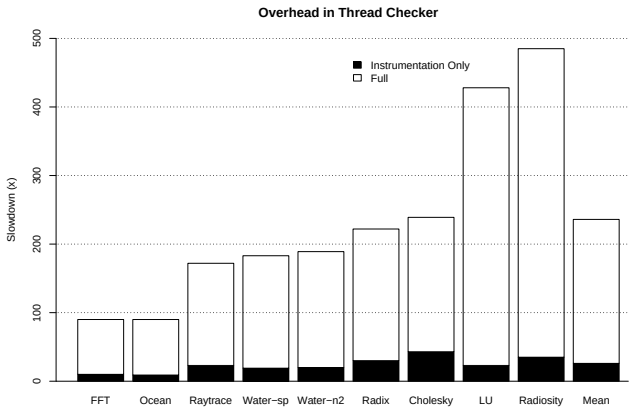

Figure 2: Time overheads in Thread Checker. The data correspond to 4-processor runs.

While each thread in an application is running, program instrumentation generates this information and puts it into a thread-private buffer. When the buffer is full or a synchronization operation is encountered, a global lock is obtained, and the vector clock algorithm runs. The buffering is done by each thread in parallel without a lock, whereas the vector clock algorithm is serialized with a global lock.

As can be seen in Figure 2, instrumentation alone causes a 22x slowdown, on average. This is found by running Thread Checker with the vector clock algorithm disabled but with instrumentation enabled. The full algorithm with instrumentation incurs a 233x slowdown, on average. Consequently, it is better to focus on optimizing the execution of the vector clock algorithm (the white section of Figure 2), rather than on reducing the instrumentation cost. Moreover, since while one thread has the global lock and is performing the vector clock algorithm, all the other threads can continue executing instructions and buffering memory reference information, it is likely that much of the instrumentation overhead is subsumed by the vector clock algorithm overhead. In other words, if we reduced the black section of Figure 2 by reducing the instrumentation cost, it is likely that the white section would grow to neutralize any change.

There are two main approaches to reducing the execution time of the vector clock algorithm. The first one is to parallelize the algorithm (i.e., parallelize the critical section). The second one is to reduce the amount of work done by the algorithm (i.e., reduce the size of the critical section). While both approaches are potentially good, the first one is likely to be harder and, in addition, not very effective when the algorithm runs with a small number of processors like in our experiments. Consequently, in this paper, we focus on the second approach. We develop a filter that can greatly speed-up the critical section and, therefore, be effective for single-processor systems as well as for large multiprocessor systems.

2.4 Our approach

In this work, we take the state machine from Eraser and combine it with two other simple filters to construct a filter for the vector clock-based algorithm. With this, we hope to eliminate the vast majority of memory references that need to be processed by the vector clock algorithm. We will eliminate those references that are not likely to be involved in data races. If our filter can filter many references without missing many data races, the algorithm can reduce the enormous overhead of data-race detection to a more manageable level.

Figure 3 shows the interactions of the various components in Thread Checker and where our filter fits in. In the example of the figure, the instrumentation in the application thread sends three references to the filter. The filter filters two of the three references and passes one reference to Thread Checker.

In the next section, we explore the design alternatives for constructing a parallel filter for Thread Checker or any other data-race detector.

3. ACCURATE & EFFICIENT FILTERING

In this section, we elaborate upon the Eraser state machine and present our filter designs.

3.1 Is an imperfect filter okay?

It is commonly held that imperfect filtering – filtering references that have even a remote chance of being involved in a data race – is inherently a bad idea, since it might cause a race condition to go undetected. The argument is that a good data-race detector should not miss any data races.

What we have found is that large, long-running applications simply exhaust the memory available in common computer systems. Thread Checker imposes a 20x memory overhead, due to storing vital debugging information for all

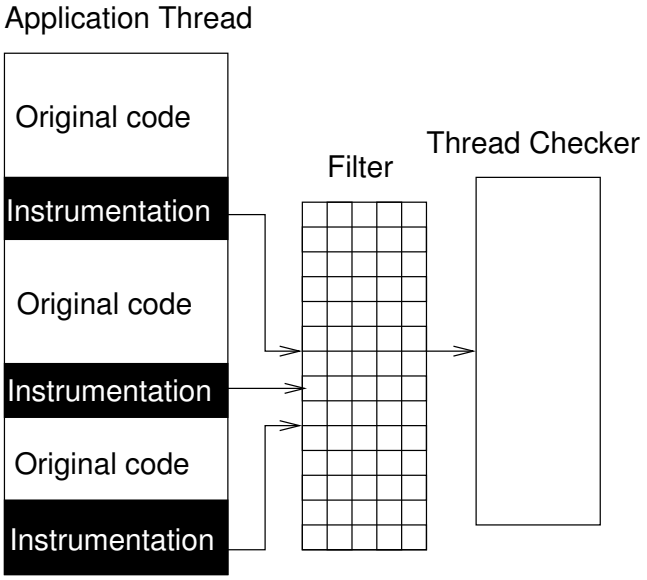

Figure 3: The proposed architecture. In this example, the instrumented thread sends 3 references to the filter. 2 are filtered, and 1 is passed to the Thread Checker vector clock algorithm.

memory references. Therefore, Thread Checker must periodically deallocate some of this information under memory pressure if Thread Checker is to be useful at all.

Further, all dynamic data-race detectors, at best, can find all the data races in one dynamic execution of an application. They cannot prove that no other data races can ever occur – e.g., code not executed in one particular run could unsafely access shared data.

Finally, users are often unwilling or unable to wait for Thread Checker to do a test run that takes 200 times as long as a normal run. Common practice is to reduce the input data set or tweak other application parameters to get an acceptable run time – thus potentially missing some of the data races that would occur in a normal run.

There is also some precedent for our viewpoint. RaceTrack [16], a promising data-race detector for the .NET managed runtime environment, uses a form of imperfect filtering to reduce overheads – namely, only accesses to a subset of array members are processed. An adversary could easily create a data race that would elude RaceTrack or our filters, but catching adversaries is not the purpose of practical data-race detectors. A practical, imperfect tool is better than an impractical, perfect tool.

In light of all this, we feel that optional memory-reference filtering, even if imperfect, is a net benefit for most users. We now elaborate on how we do this.

3.2 Filtering useless references

The vast majority of memory references will not be involved in a data race. Further, it has been our experience that most data races that occur in a program recur many times. Thus, it seems prudent to filter unneeded references in the instrumentation and buffering stages before they reach the slow vector clock-based data-race detection algorithm.

Variables that are only used by one thread cannot be involved in a data race. Therefore, stack variables cannot be involved in a data race in most programs. Moreover, variables residing in the heap or the data segment that are only used by one thread should be filtered. In addition, variables that are only read by multiple threads should be filtered as well. Finally, variables repeatedly accessed within or between critical sections need only be passed through once.

3.3 State machine choices

We propose to use three filters to filter away references unlikely to be involved in data races.

3.3.1 Stack filter

First, we filter away stack references. This is a low-overhead filter – the address of the memory reference is compared against the stack base and limit addresses, and then filtered or passed through. Code in Thread Checker checks if one thread accesses another's stack. In this case, the user is notified and this filter can be disabled.

For most applications – those that do not have cross-thread stack accesses – this filter cannot cause any data races to be lost and is very efficient.

3.3.2 Duplicate filter

Next, we filter duplicate references in segments. The first load and store references to a variable in each segment by a thread are passed through, while subsequent loads and stores to the same variable are filtered.

The list of addresses already accessed in the current segment is kept in a simple per-thread table. Further, addresses involved in prior data races are marked in the table, and all future accesses to those addresses will not be filtered, whether duplicate or not. The actual operation is discussed later.

This filter can only cause Thread Checker to lose *duplicate data races*. A duplicate data race is a data race that involves the same variable, threads, and thread segments as an already-detected data race. Losing a duplicate data race may make the data-race causing bug more difficult to diagnose, but will not cause any synchronization bugs that would otherwise be detected to lie undetected.

We have found duplicate data races to be very common. Consider a critical section in which a shared variable is read, then written. If this critical section races with an unsynchronized write to the shared variable, both the read and the write will be involved in duplicate data races, but only one data race needs to be reported.

3.3.3 FSM filter

The stack and duplicate filters catch most of the useless memory references most of the time, but we have found some cases in which it is useful to add a third, final filter.

Our approach for this filter is to leverage the Eraser state machine used to reduce false positives in the Eraser algorithm, and apply it to Thread Checker for a different purpose – filtering memory references.

In the following section, we will describe the specific organization of our filter, but for now let us first consider the state machine behind it.

3.3.3.1 Eraser state machine.

In the Eraser state machine, each variable starts in the Uninitialized state. As show in Figure 1, after the first ac-

cess, the variable enters the Private state. If the same thread continues to access the variable, the variable remains in the Private state.

If another thread reads the variable, it enters the Shared Read-Only state. In the Shared Read-Only state, locksets are intersected and saved, but data races are not reported. If another thread writes to the variable, it enters the Shared Read-Write state. It is in this state that data races are reported.

3.3.3.2 *Our state machine.*

We base our filter upon the Eraser state machine. We filter references in the Private state and in the Shared Read-Only state. We pass through references in the Shared Read-Write state. The shared variables that are read and written and could be involved in data races will quickly transition to the Shared Read-Write state and not be filtered. Variables that are privately used by one thread will stay in the Private, filtered state. Variables that are shared but only read will stay in the Shared Read-Only, filtered state.

Further, we also pass through *the initial references that cause a state transition* from the Uninitialized state to the Private state, and from the Private state to the Shared Read-Only state. This has a minimal impact on filter rate, but is valuable in detecting data races that involve some initial references to a variable.

4. IMPLEMENTATION ISSUES

In this section, we describe the organizations of our memory reference filters.

4.1 Stack filter

The stack filter is the simplest filter and has the lowest overhead, so we apply it first. It compares the memory reference address with the stack base and limit addresses. If it falls within the range, the reference is filtered; otherwise, it is passed through.

4.2 Duplicate filter

This filter requires a table lookup, and therefore is slower than the stack filter, so it is applied next. Each thread maintains its own duplicate filter table. We chose to use a 16k-entry filter table organized as a direct-mapped cache. Each entry has several fields:

- The address of the reference
- The size of the access (byte, word, double-word)
- The type of access (read or write)
- The segment ID

The 16k-entry table requires a 14-bit index. The lowest 2 bits of the full virtual address of the reference are XORed with the next-lowest 14 bits to form an index. This is because most accesses are at the word or double-word (floating-point) granularity and, therefore, the lowest 2 bits are less useful.

If the address, size, type, and segment ID match, then an identical reference has already been processed in this segment and the reference can be safely filtered away. If the address matches but the other fields do not, the entry is augmented and the reference is passed through. If the address does not match, the address and the fields are filled in, and the reference is, again, passed through.

Duplicate references are often generated by different static instructions. It can be useful to the programmer to see the different lines of source code involved in duplicate races. Therefore, we have added a feedback loop from the data-race detector to the duplicate filter. It uses an extra subfield in the access-type field to mark the entry as having been involved in a data race. Subsequent matches to this entry will never be filtered.

4.3 FSM filter

This filter uses a table shared by all threads with some synchronization, and, therefore, is the slowest. It is applied last.

The filter requires two fields for each variable:

- The state the variable is in. This uses 2 bits, since there are 4 states – Uninitialized, Private, Shared Read-Only, and Shared Read-Write.
- The thread ID for the Private state. We have chosen 14 bits for this, such that each entry requires two bytes of storage.

An efficient implementation must work with memory addresses and not variable names. The filter, as input, takes a 32-bit memory address and the size of the access. One possibility is to filter at the byte level. In this case, for an 8-byte double-precision floating-point access, 8 filter entries would need to be checked. However, such a solution would be slow and would consume too much memory.

Suppose, instead, that we choose to filter at the 32-bit (4-byte) block level:

- When there are variables used that are smaller than the chosen block size, such as bytes, there will be aliasing within a block. If one byte within a 4-byte word is private to one thread, and another byte in the word is private to another thread, our filter may transition to the Shared Read-Write state and pass through references that are not necessary. We are underfiltering.
- When there are variables used that are larger than the chosen block size, such as double-precision floating-point numbers, we will have to access two entries to ensure that we do not overfilter. This will ensure that we do not filter the case when, for example, there is a race between a store to the second half of a double and a load to the entire double.

Intuition and experimentation has shown that a 4-byte block size is optimal.

We also found that a 4-way set associative table with LRU had the best tradeoff between filtering rate and overhead. We chose to use a 1M-entry table, which covers 4 MB of application data at a time. The table is indexed and tagged like a set-associative cache, and has an actual footprint of about 4 MB.

The common path through the filter is when the most-recently used entry in the set matches the referenced address, and no updates are required to the entry. In this case, we do not have to update the table and do not have to do any synchronization operations.

If there is not a match on the most-recently used entry or the most-recently used entry requires an update, an atomic

fetch-and-swap operation is used to write a special value to the LRU variable, locking the set. Then the entry is updated, (if necessary). Finally, the LRU variable is updated, implicitly unlocking the set.

5. EXPERIMENTAL SETUP

We use the SPLASH-2 applications in our evaluation [14]. All applications were run with 4 threads on 4 processors, using the standard data set. Statistics are collected during the entire run.

The performance measurements are obtained on a 4-way 2.5 GHz Pentium 4 workstation. Filtering statistics are collected by running each application three times for each configuration; performance results are collected by running each application nine times for each configuration. All figures show 95% confidence limit segments calculated using the Student's t-statistic. However, it is difficult to perceive such confidence limit segments in the graphs showing filtering rates, as filtering rates vary very little from run to run.

Finally, each application is run in Thread Checker with and without our filters to see how data race detection is affected. We are interested in determining whether or not using the filters reduces the number of data races reported to the user. We compare the number of data-race bugs reported with and without the filters.

A data-race causing bug usually results in many reported data races. Thread Checker collapses all races that involve the same pair of lines of code and are of the same type (read-write, write-write, or write-read) into one diagnostic message. A diagnostic message from Thread Checker reports the pair of lines of code, the call stack, and the type of data race.

6. EVALUATION

In our evaluation, we examine the filtering effectiveness of our technique, its performance impact, and its impact on data-race detection capability.

6.1 Filtering effectiveness

Figure 4 shows the average filtering rate (i.e., the fraction of references that are eliminated) for each application for different filters: the stack filter; the stack and duplicate filters together; and the stack, duplicate, and FSM filters together.

From the figure, we see that the stack filter is clearly effective. On average, it filters out over 50% of all references. Further, it has a very low implementation cost and, in most applications, cannot cause Thread Checker to miss any race conditions.

When the duplicate filter is used after the stack filter, the average filtering rate increases to 90%, and when all three filters are used, the average filtering rate is 98%. It is questionable whether an additional 8% filtering rate justifies the use of the FSM filter – the FSM filter is the only filter of the three that can cause data races to be missed. However, in some applications, such as *Ocean*, it is indeed quite beneficial: it increases the filtering rate from 68% to 97%.

Figure 5 shows the incremental filtering obtained with each filtering scheme. The first bar shows the percentage of references filtered by the stack filter; the second bar shows the percentage of references passed through by the stack filter that are filtered by the duplicate filter, and the third bar

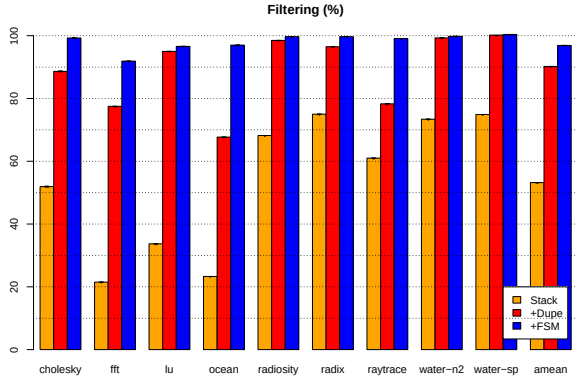

Figure 4: Filtering effectiveness of different filter combinations.

shows the percentage of references passed through by the stack and duplicate filter that are filtered by the FSM filter.

For some applications, such as *Water-n2* and *Water-sp*, the FSM filter does not filter much, but these applications already have nearly 100% filtering rates, so it is of little consequence. For other applications, such as *Ocean* and *Raytrace*, the FSM filter filters 90% or more of references, and is very useful, increasing combined filtering rates from 70-80% to nearly 100%.

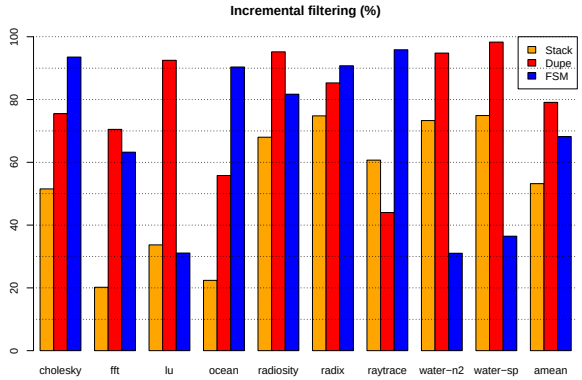

Figure 5: Incremental filtering effectiveness.

6.2 Performance

Figure 6 shows the Thread Checker speedups obtained with filtering, normalized to the base case of not using any filtering. On average, stack filtering provides a 1.5x speedup, stack and duplicate filtering provides a 2.6x speedup, and the combination of all three filters provides a 3.3x speedup. The three filters together speed-up individual applications from 2.2x to 5.5x.

Thread Checker has many sources of overhead. Logically, if we obtain an over-2x speedup with filtering, then the data-race detection algorithm must be the largest contributor. But with 90%+ filtering rates, we do not reap a

10x speedup. This is because there are many other overheads in Thread Checker that become more dominant as the overhead of the date-race detection algorithm becomes less dominant. These include program instrumentation, call stack generation, deadlock detection, and other fixed costs.

Nonetheless, an average speedup of 3.3x and up to 5.5x can be obtained using a straightforward series of filters.

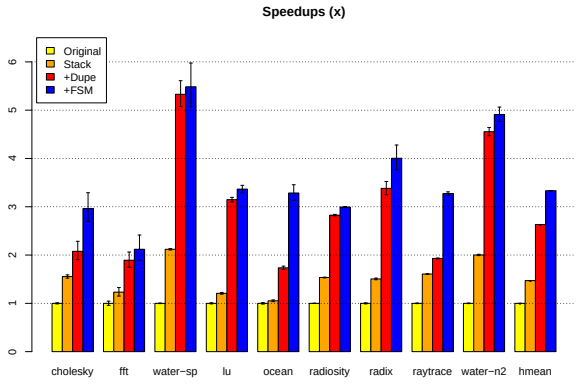

Figure 6: Thread Checker speedups obtained with filtering.

6.3 Data-race detection

Thread Checker detects 12 data races among the SPLASH-2 applications. The one in *Ocean* can be described as truly benign. Most of the others will be benign under some memory models and racy under others. Most of these races were due to hand-coded synchronization routines which did not use memory fences or library synchronization methods.

The results when Thread Checker is used with all three filters combined are summarized in Table 1. For every application we tested, Thread Checker detected identical sets of races whether or not filtering was enabled.

The benign data race in *Ocean* was a redundant assignment, wherein a shared variable was initialized by every thread to the same value and never updated again. The applications were coded such that it was simple for a programmer to see that this is what was happening, but a data race detector, even with inter-procedural analysis, could never prove it.

The table also shows that, without filtering, the overhead of using Thread Checker varies from a slowdown of 90x to 485x, with an arithmetic mean of 233x. With our filter, slowdowns are reduced to 28-163x, with an arithmetic mean of 69x.

We also see that our filters can filter 98% of references in SPLASH-2 on average and miss no malignant data races. Still, since there is a small possibility that a data race could lie undetected when the FSM filter is used, we would recommend that users perform a final test run with that filter disabled.

7. RELATED WORK

As mentioned early in the paper, there are two classes of data-race detection tools. Tools in the first class, such as

Application	# Races (Filter/ No Filter)	Filter Rate (%)	Original Overhead (Times)	Overhead w/ Filter (Times)
Cholesky	1/1	99	239	82
FFT	0/0	92	90	41
LU	1/1	97	428	128
Ocean	1/1	97	90	28
Radiosity	5/5	99	485	163
Radix	2/2	99	222	56
Raytrace	2/2	98	172	53
Water-n^2	0/0	99	189	39
Water-sp	0/0	99	183	34
Average	—	98	233	69

Table 1: Characterizing the impact of the three filters combined.

Eraser [13], are based upon detecting violations of a locking discipline, where a locking discipline refers to a consistent set of locks used to access a given shared variable during multi-threaded execution.

The second class, such as Thread Checker [5], are based upon vector clocks. Vector clocks create an ordering among thread segments based upon synchronization events, such as locks, thread creation, semaphores and barriers. Vector clock-based data-race detectors detect when two conflicting accesses are made to a shared variable by two unordered thread segments.

There is much work based on vector-clock algorithms [3, 11, 12, 15] for debugging and deterministic replay. One of the seminal works on using vectors clocks for race detection is by Netzer and Miller [9].

Sequential data race detectors, such as [1] and [8], only guarantee finding the first race in an application. Its usage model requires its users to fix the first data race in an application before finding the next. Many users may not be willing to fix benign data races. Others may want to solve simpler data races first, and then move on to the more complex ones. Finally, programmers may not have access to the source code for all the libraries they use; they simply are unable to fix the first data race if it occurs in library code. In general, however, these algorithms have lower overhead than Thread Checker. An interesting extension is Nondeterminator-3 [6], a sequential data-race detector which runs in parallel.

Recently, there has been some work on combining lockset and vector-clock based algorithms. In [10], locksets are used to detect data races in Java applications, and vector clocks are used to eliminate many false positives. RaceTrack [16] uses locksets to find potential data races and vector clocks to see if the potentially racing accesses are properly ordered or not.

There is little work relating to front-end filtering for data-race detection algorithms. In [2], a cache similar to our duplicate filter is used to improve performance. Our work evaluates three different kinds of filters and characterizes their effectiveness. We believe that our filters could be applied to many other data-race detectors to improve their performance.

8. CONCLUSIONS AND FUTURE WORK

Data-race detectors such as the Intel Thread Checker are useful in helping programmers debug difficult race condi-

tions. Unfortunately, as we have seen, there can be immense slowdowns of up to 485x, with an average slowdown of 233x.

In this work, we develop a stack, duplicate, and FSM filter which, on average, filter 98% of all references and do not miss any of the 12 data races Thread Checkers finds in the SPLASH-2 benchmarks. This effects an average 3.3x speedup, because other overhead sources become the limiting factors.

We believe that the 69x slowdown remaining after using our filters is still too much. We have several ideas on how to improve the filters. One idea is to use feedback information from the vector clock algorithm which might allow us to safely downgrade some entries in the FSM filter from the Shared Read-Write state to the Private or Shared Read-Only state. There are also some optimizations that we have considered that might reduce the overhead of the FSM filter without impacting its filtering effectiveness much. However, even close to 100% filtering will not provide much more speedup.

Much work can be done to improve the other overhead sources in Thread Checker, such as the instrumentation, the call-stack generation, and the serialization of the core algorithm. Our work has substantially widened the biggest bottleneck to Thread Checker performance, without which other optimizations would have little benefit.

9. REFERENCES

[1] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Symposium on Parallel Algorithms and Architectures*, 1998.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *International Symposium on Programming Language Design and Implementation*, 2002.

[3] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[4] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[5] Intel Corporation. Intel Thread Checker. http://www.intel.com/support/performancetools/threadchecker.

[6] T. C. Karunaratna. Nondeterminator-3: A provably good data-race detector which runs in parallel. *MIT CSAIL report*, 2005.

[7] F. Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[8] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing*, 1991.

[9] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. *International Symposium on Principles and Practice of Parallel Programming*, 26(7):133–144, 1991.

[10] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *International Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[11] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, pages 110–121, 2003.

[12] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, 1995.

[15] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, pages 122–135, 2003.

[16] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *International Symposium on Operating Systems and Principles*, 2005.