Parallelizing Data Race Detection

Benjamin Wester[†*]

[†]Facebook
bwester@fb.com

David Devecsery[*] Peter M. Chen[*]
Jason Flinn[*] Satish Narayanasamy[*]

[*]University of Michigan
{ddevec,pmchen,jflinn,nsatish}@umich.edu

Abstract

Detecting data races in multithreaded programs is a crucial part of debugging such programs, but traditional data race detectors are too slow to use routinely. This paper shows how to speed up race detection by spreading the work across multiple cores. Our strategy relies on *uniparallelism*, which executes time intervals of a program (called *epochs*) in parallel to provide scalability, but executes all threads from a single epoch on a single core to eliminate locking overhead. We use several techniques to make parallelization effective: dividing race detection into three phases, predicting a subset of the analysis state, eliminating sequential work via transitive reduction, and reducing the work needed to maintain multiple versions of analysis via factorization. We demonstrate our strategy by parallelizing a happens-before detector and a lockset-based detector. We find that uniparallelism can significantly speed up data race detection. With 4× the number of cores as the original application, our strategy speeds up the median execution time by 4.4× for a happens-before detector and 3.3× for a lockset race detector. Even on the same number of cores as the conventional detectors, the ability for uniparallelism to elide analysis locks allows it to reduce the median overhead by 13% for a happens-before detector and 8% for a lockset detector.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design; D.4.8 [*Operating Systems*]: Performance

General Terms Design, Performance

Keywords Data race detection, Uniparallelism

1. Introduction

The prevalence of multicore processors has led to an increased use of multithreaded programs. In such programs, several threads of execution share a single address space and coordinate their accesses to shared variables via synchronization operations, such as locks and condition variables.

Multithreaded programs are subject to a variety of concurrency bugs, of which a particularly pernicious type are *data races* [5]. A data race occurs when two threads access the same memory location without being ordered by a synchronization operation and at least one access is a write. Data races tend to be nondeterministic and difficult to debug, and they can lead to severe problems at runtime [20, 31]. Even enumerating the possible behaviors of programs with data races is difficult—the C and C++ language specifications explicitly leave the behavior of such programs undefined [4], and the Java specification for such programs is extremely complex and currently has known bugs [41].

To address this problem, researchers have developed tools to detect data races in multithreaded programs. We focus in this paper on dynamic data race detectors; static race detectors also exist but currently suffer from an excess of false positives. During development, data race detectors can help programmers find and eliminate data races. During production, data race detectors can try to avoid racy executions or can stop the program before it generates incorrect or arbitrary output.

The main problem preventing the widespread use of dynamic data race detectors is their high overhead, especially for tools that can operate on arbitrary binaries. For example, Intel Inspector incurs an average of 233× slowdown for the SPLASH-2 benchmarks [38], and Google's ThreadSanitizer (happens-before mode) incurs an average of 29× slowdown for a series of unit tests [40]. Even research systems that operate on managed code are slow, e.g., FastTrack [15], a state-of-the-art, precise race detector, imposes an average of 8.5× slowdown on the execution time of Java programs. The high overhead of these tools arises from needing to instrument and analyze each potentially-racy memory operation.

Several strategies have been explored to reduce the overhead of data race detection. One approach is to rely on custom hardware [11, 24, 33, 51], but such support is not yet available on commodity processors. Another approach is to tolerate false negatives (i.e., miss detecting some data races), such as by sampling only a portion of the execution [6, 22] or data space [13], by refining the granularity of detection at the cost of missing the first race on a variable [50], or by detecting the outcome of races rather than the race itself [43]. A third approach is to tolerate false positives (i.e., report some events that are not data races), such as by increasing the granularity of detection [46].

This paper presents a new strategy for accelerating a data race detector by enabling it to run in parallel across multiple cores. Surprisingly, not only does our method of parallelization scale well with increasing cores, it also usually reduces the overhead of data race detection even when run on the original number of cores. Our strategy relies on *uniparallelism* [44], which executes time intervals of a program (called *epochs*) in parallel, but executes all threads from a single epoch on a single core. To enable parallel execution of epochs, we run a lightweight program replica that conducts little race analysis, and we use this copy to predict future application and partial analysis state. Executing epochs across cores increases performance via parallelism. Executing all threads from a single epoch on one core increases performance by eliminating the need for the analysis code to acquire and release locks.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

We demonstrate how to adapt two traditional data race detectors (happens-before and lockset) to run in a uniparallel environment. Each detector must be restructured into three phases: the first phase runs epochs in sequence and predicts a subset of the analysis state, the second phase runs epochs in parallel and performs the bulk of the analysis, and the third phase merges together the results of the parallel phase. The second phase detects races that occur within a particular epoch, and the third phase detects races that occur across epochs.

We find that uniparallelism can significantly speed up data race detection. With 4× the number of cores as the original application, our strategy speeds up the median execution time by 4.4× for a happens-before detector and 3.3× for a lockset race detector. Even on the same number of cores as the conventional detectors, the ability for uniparallelism to elide analysis locks allows it to reduce the median overhead by 13% for a happens-before detector and 8% for a lockset detector.

In this paper, we make several novel contributions to the area of race detection and parallel program analysis. First, we show how running multiple epochs in parallel can parallelize race detection, even though detecting races for one epoch depends on the analysis done in prior epochs. We show how race detection in general can be restructured into three phases: one that predicts a subset of the analysis state, a second that runs the bulk of the analysis, and a third that resolves the symbolic expressions that were logged in the second phase. Second, we show how two classic race detectors (based on happens-before and lockset) can be reimplemented in this architecture. In particular, we show how to partition the work of those race detectors effectively into phases, reduce the amount of deferred work through transitive reduction, and reduce the amount of work needed to deal with unknown initial states through lockset factorization. Third, we show how uniparallelism reduces the overhead of fine-grained analyses such as race detection by eliminating locking overhead. Finally, we demonstrate that parallel race detection scales on commodity hardware.

2. Parallel race detection

It is challenging to parallelize fine-grained program analyses such as race detection. Typically, the analysis is done synchronously with the program, so the speeds of the program and analysis are linked. In addition, some race detectors have strong dependencies between instructions, i.e., the analysis of an instruction depends upon the analysis of prior instructions.

One way to parallelize a race detector is to run the application on more cores and run the race detector along with the scaled application. There are three problems with this approach. First, many applications cannot easily take advantage of more cores. Second, as we show in Section 6, applications that can adapt to more cores do not necessarily scale well, especially when run with race detection. Finally, a developer may want to debug the original application rather than an application that has been scaled to use more cores.

Another way to parallelize race detection is to instrument an application to log memory accesses and then feed the trace to multiple threads that perform data race detection. The problem with this approach is that the memory access logger must monitor every memory operation, and this may become a performance bottleneck without hardware support [7].

Our goal is to parallelize race detection in a way that does not suffer from these problems. Specifically, we would like to parallelize race detection without modifying the application, and we would like the parallel race detector to scale well to more cores even if the original application does not scale well. To achieve this goal, we use a previously proposed technique called uniparallelism [44].

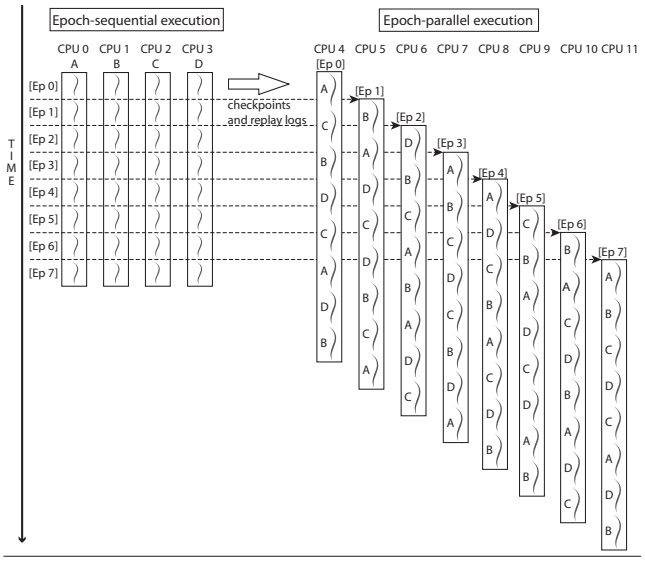


Figure 1. Uniparallelism

We first present an overview of uniparallelism in the next section; we then describe how we apply it to race detection.

2.1 Background: uniparallelism

Uniparallelism operates on unmodified applications. It achieves parallelism via a technique called epoch parallelism [29] (also called master/slave speculative parallelization [52] and predictor/executor approach [42]).

In epoch parallelism, the application is divided into distinct time intervals, called *epochs*. The goal of epoch parallelism is to run these epochs in pipelined fashion, where later epochs start before earlier ones finish. Epochs are isolated in their own address space, and the program's semantics are equivalent to a sequential execution: e.g., writes to local memory or files in one epoch are visible to epochs that logically occur later. With more cores available, the pipeline can be deeper and more epochs can run in parallel.

An immediate question that arises is: "How can a later epoch start before an earlier epoch finishes?" Starting epochs early in this manner depends on *predicting* the starting state of that epoch. The predicted state includes the architectural state of the process (the entire address space and all thread registers) as well as the relevant system state (e.g., so that file accesses are consistent). To generate these predictions, epoch parallelism runs a second execution of the application. This second execution runs normally, executing epochs sequentially. In this paper, we refer to this execution as the *epoch-sequential execution*, in contrast with the pipelined execution, which we refer to as the *epoch-parallel execution*. Figure 1 depicts these two executions.

Three properties must be true for epoch parallelism to work. First, the epoch-sequential execution must run ahead of the epoch-parallel execution, so it can predict the starting state of future epochs. In our work, we provide this property by performing most of the work of race detection only in the epoch-parallel execution. Since the work performed for race detection is much greater than the work to execute the original program, the epoch-sequential execution quickly outpaces the epoch-parallel execution.

Second, the state predictions generated by the epoch-sequential execution must *usually* be correct. While incorrect predictions do not affect correctness, they do reduce performance. If the predicted ending state for an epoch (generated by the epoch-sequential execution) does not match the actual ending state (generated by the

epoch-parallel execution), the system must flush the pipeline of any epochs that depend on the incorrect prediction, and the epoch-sequential execution must restart from the last committed state (this is called a *rollback*). At any given time, the epoch that is currently the oldest in the pipeline is non-speculative and can release output; all other epochs are speculative and will be discarded in a rollback. Each rollback causes the pipeline to be flushed, which reduces parallelism and wastes the CPU time spent on the flushed epochs. Epoch parallelism ensures the accuracy of the predicted state in the common case via *online replay* [19]. The epoch-sequential execution logs most sources of nondeterminism, and the logged values are replayed during the epoch-parallel execution. Logged events include all synchronization operations (e.g., all `pthread` operations) and input received from system calls (e.g., disk and network I/O). All synchronization operations are replayed in a way that obeys the happens-before partial order of synchronization objects observed during recording.

Although the order of shared-memory accesses can also be nondeterministic, these accesses are too expensive to log and replay. However, since both executions use the same happens-before order of synchronization objects, shared-memory accesses can cause the epoch-sequential and epoch-parallel executions to diverge only when there is a data race [35]. In fact, many data races may not cause rollbacks because they do not affect the state of the program [25]. Programs with frequent data races may cause frequent rollbacks, but the speed of race detection is not critical for such programs because even a slow race detector can find races easily if they occur often.

Third, the system must detect when the state predicted by the epoch-sequential execution is incorrect. Previous epoch-parallel systems detect incorrect predictions by comparing the memory state of the two executions. However, we observe that a useful synergy exists when epoch parallelism is used to parallelize race detection. Since the prediction can only be incorrect when a prior epoch contains a data race, no memory comparison is required if a sound data race detector has determined that all prior epochs are race free.

Putting it all together, epoch parallelism runs the application twice: an epoch-sequential execution to generate predictions of future epoch states, and an epoch-parallel execution to conduct the race analysis. Although the epoch-parallel execution takes place in an unusual, pipelined manner, it is nonetheless a legitimate, real execution of the application. Its computation and output are identical to an execution in which the epochs are stitched together sequentially, because the ending state of an epoch matches the starting state of the next epoch (otherwise the system will rollback), and output is released only as epochs commit.

Uniparallelism is a further refinement of epoch parallelism in which all threads for a given epoch execute on the same core via timeslicing. Threads are scheduled onto the core non-preemptively, switching only when needed to follow the happens-before order determined by the epoch-sequential execution. Uniparallelism was motivated by the observation that some properties, such as deterministic record and replay [44] or the enforcement of specific thread orders [43], are much less expensive to achieve on a single processor than they are on a multiprocessor. Section 2.3 explains how uniparallelism can increase the efficiency of race detection, even when run on the same number of cores as a conventional detector.

2.2 Parallelizing the analysis

While uniparallelism increases the degree of parallelism for an application, it is not at all obvious how one can use epoch parallelism to parallelize the work of detecting races for that application. A key requirement of uniparallelism is that the epoch-sequential execution must run faster than the epoch-parallel execution, and this occurs only because the epoch-sequential execution is not encumbered by the work of detecting races. This means that the state needed to detect races (*analysis state*) is not generated or maintained by the epoch-sequential execution and hence is not part of the predicted state used to start epochs in the epoch-parallel execution (*application state*). For example, for happens-before race detectors, the epoch-sequential execution should not maintain the vector clocks for each variable, and for lockset-based race detectors, the epoch-sequential execution should not maintain the locksets for each variable. Otherwise, it would run as slowly as the epoch-parallel execution and would no longer be able to run ahead to predict states for future epochs.

Our strategy for parallelizing the analysis of the application is to divide it into three phases: the epoch-sequential phase, the epoch-parallel phase, and the commit phase. The bulk of the work is performed in the epoch-parallel phase; the epoch-sequential phase predicts a subset of the analysis state used in the epoch-parallel phase; and the commit phase carries out work that cannot be performed by the epoch-parallel phase because the epoch-parallel phase starts with incomplete state. This section describes these phases independently of the race detection algorithm; Sections 3 and 4 detail how we divide the work of specific race detectors into these phases.

2.2.1 Epoch-sequential phase

The epoch-sequential phase of race detection runs in the epoch-sequential execution. Its purpose is to predict a *subset* of the analysis state at the beginning of each epoch. The epoch-parallel phase can then use this predicted state when it analyzes each epoch in the epoch-parallel execution. Predicting a subset of the analysis state is similar to how the epoch-sequential execution predicts the state of the application at the beginning of each epoch. As with application state, the subset of analysis state is generated twice: once in the epoch-sequential phase to form the prediction, and another time in the epoch-parallel phase to maintain this state and to verify the prediction for the next epoch.

The work done in the epoch-sequential phase must meet three requirements. First, the subset of analysis state must be self-contained: updating the state subset must not depend on analysis state outside the subset.

Second, the work done in the epoch-sequential phase must be lightweight so that the epoch-sequential execution can continue to run ahead of the pipelined epoch-parallel execution. All events in the application that modify the analysis state in the subset must be instrumented and analyzed during the epoch-sequential phase, so the subset should ideally be chosen so that most instructions do not require it to be updated.

Third, the predicted analysis state should usually match the analysis state generated in the epoch-parallel execution, since mispredicted values cause rollbacks and thereby reduce performance. Recall that we guarantee that the application states generated in the epoch-sequential and epoch-parallel executions match for race-free programs by logging and replaying all sources of non-determinism except data races. Similarly, to avoid mispredictions in race-free programs due to mismatches in predicted analysis state, in our designs we ensure that all executions consistent with a happens-before order of synchronization operations will generate the same values for the subset of analysis state.

2.2.2 Epoch-parallel phase

The epoch-parallel phase of race detection runs as part of each epoch in the epoch-parallel execution. This work is pipelined along with the epochs and hence scales with increasing cores. To achieve the greatest scalability, most race detection work should occur in this phase.

Unfortunately, all the work of race detection cannot take place in the epoch-parallel phase because detecting races in one epoch depends on the analysis of prior epochs. Race detection algorithms are stateful: the analysis state gathered by analyzing prior epochs is required to correctly detect all races involving the current epoch. As a trivial example, identifying a race that spans epochs requires analysis state from both epochs. Further, even if the racing pair of accesses occurs within the same epoch, a race detector may require analysis state from prior epochs to identify the pair as racing. For example, with a lockset-based race detector, analyzing a variable access depends on knowing the set of locks that are held by a thread, which depends on the synchronization actions of prior epochs.

Some of the analysis state needed by the epoch-parallel phase is predicted by the epoch-sequential phase. However, state outside this subset is not known when the epoch starts. When the epoch-parallel phase needs to evaluate an expression that contains unknown state, it logs a symbolic form of the expression and defers the evaluation of this expression until the commit phase.

2.2.3 Commit phase

Because the epoch-sequential phase predicts only a subset of the analysis state, the epoch-parallel phase of race detection starts each epoch with incomplete information and may thus be unable to perform all the work of race detection. For example, it is too slow for the epoch-sequential phase to update state on each variable access, so the epoch-parallel phase will not be able to identify races that depend on the analysis state for a variable at the beginning of an epoch. To finish the work of race detection that was unable to be completed in the epoch-parallel phase, we add a final phase of race detection, which we call the commit phase.

The commit phase for an epoch E occurs after all prior epochs have been committed (i.e., commit phases execute sequentially). At this time, the complete analysis state at the beginning of epoch E is known, having been generated by the commit phases of the epochs prior to E. The commit phase can then use this complete analysis state to resolve all unclassified events from the epoch and to generate the complete analysis state for the end of epoch E.

Because the commit phase occurs after the epoch has finished, races detected during this phase are detected later than they would have been detected by a conventional on-the-fly detector. For some uses of race detection, this delay has no effect. For example, race detection could be used on a production system to provide fail-stop behavior for races, and the delay has no effect because speculative output is not released until the commit phase completes. However, for debugging, developers may want to recover the program's state at the time of the access rather than at the end of the epoch. For such a use case, the epoch could be replayed to the racing instruction via the same log used to synchronize the epoch-parallel execution with the epoch-sequential execution [35].

2.3 Eliding analysis locks

In addition to scaling race detection to more cores, uniparallelism can reduce the overhead of race detection even when using the same number of cores. To understand how, one must first consider how the code used to detect races is itself multithreaded. Race detectors read and write analysis state, and this state is shared across threads when the detector is analyzing a multithreaded program. Even if the original program contains no races, the race detector must be written carefully to avoid introducing races on this analysis state. For example, two threads may concurrently read an application variable without causing a data race, but the analysis code for these two events may update analysis state (e.g., a lockset) and must therefore not execute concurrently. As another example, analyzing accesses to separate variables may result in updating the same analysis state, and these updates must not execute concurrently.

A conventional way to provide atomicity for parallel analysis code is to use locks. The epoch-sequential phase of analysis uses this strategy to provide atomicity. However, acquiring and releasing locks can add significant overhead [10], especially when analyzing high-frequency events such as memory accesses.

An advantage of uniparallelism is that it guarantees atomicity for the race detection code that analyzes program events without needing the code to acquire and release locks. Only a single thread from each epoch executes at a time, and different epochs do not share analysis state (until the commit phase, which is run sequentially). Therefore, locks can easily be elided by restricting the code points at which context switches can occur (i.e., as is done in uniparallelism where such switches occur only during synchronization operations). Section 6 shows that eliminating the lock acquisitions and releases for the epoch-parallel phase can reduce the total CPU time needed for the epoch-parallel phase and thereby reduce the overhead of race detection.

In summary, race detection benefits from uniparallelism in two distinct ways. First, epoch parallelism allows race detection to scale well with increasing numbers of cores, even if the underlying program itself does not scale. Second, the refinement of uniparallelism elides locks around analysis code to decrease the cost of race detection for equivalent numbers of cores.

2.4 Synergy between race detection and uniparallelism

It is possible to completely separate the design of race detection from that of uniparallelism, but doing so ignores the synergies that are made possible by co-designing the two. For example, one can implement any race detector by dividing it into two parts: logging of events and analysis of events. One could then run this detector in a uniparallel style without modification: the epoch-parallel phase would log the events and the commit phase would analyze the logged events. While this implementation is amenable to all detectors, it is not scalable—the bulk of the work, i.e., the analysis of those operations, would be done sequentially in the commit phase. For performance to scale well with increasing cores, we must move more of the work into the epoch-parallel phase. Sections 3 and 4 show how we adapt the race detection algorithm for use in a uniparallel execution style.

Just as adapting a race detector to uniparallelism has benefits, so too does adapting the uniparallel execution system to the race detector. Uniparallelism depends on predicting application state (and analysis state, when used for program analysis). These predictions are guaranteed to be correct if there are no data races, since the uniparallel execution replays the happens-before relationship from the epoch-sequential execution. Prior uniparallel systems had to check these predictions by comparing the memory state of the epoch-sequential and epoch-parallel executions [44]. However, no memory comparison is needed when uniparallelism is used to parallelize a sound race detector because the race detector itself can detect potential mispredictions. We take advantage of this observation when parallelizing a sound, happens-before race detector by eliding this memory comparison.

3. Parallel happens-before

There are two main approaches used by race detectors: (1) detect when conflicting memory accesses are not ordered by the happens-before relationship [18] and (2) detect when a program violates a particular locking discipline [39]. This section shows how to parallelize a happens-before race detection algorithm. We examine how to parallelize FastTrack [15], a state-of-the-art precise detection algorithm, as a representative example of happens-before detectors (several other detection algorithms are closely re-

lated [3, 6, 11, 16, 32]). Section 4 describes how we parallelize the other approach to race detection by examining a locking-discipline detector.

Overview: We parallelize FastTrack by instrumenting only synchronization operations, such as lock/unlock and fork/join, in the epoch-sequential phase so that their happens-before order can be predicted. Each epoch in the thread-parallel phase additionally instruments memory accesses (i.e., it runs the full FastTrack algorithm), though some race checks cannot be evaluated for lack of data from earlier epochs. Those checks are logged for later evaluation in the commit phase.

3.1 Conventional detection

Happens-before race detectors find accesses (at least one of which is a write) to the same memory location that are not ordered with respect to the happens-before relation of program actions [18]. The happens-before relation is the least-restrictive partial order that obeys the following rules: (1) An event in one thread *happens before* a subsequent event in the same thread; and (2) a release of a synchronization object (e.g., lock) *happens before* a subsequent acquisition of that synchronization object. The happens-before relation is a partial order and is therefore transitive. It can be used with any synchronization operation such as locks, condition variables, barriers, atomic accesses, etc.. Creating a new thread can be modeled as the parent thread releasing a synchronization object and the new thread acquiring that object.

Race detectors process memory accesses and synchronization events in some serial order that is consistent with the happens-before relation; this serial order is usually determined by feeding events to the race detector as they occur in real time. When discussing the ordering of events, we specifically note when we are referring to the happens-before order. Otherwise, we are referring to the serial order in which events are processed.

As is common for happens-before race detectors, FastTrack guarantees detection only of the first race on each variable [3, 8, 15, 16, 32]. Detecting the first race on each variable is sufficient for common uses of race detection, e.g., treating data races as exceptions in production [12, 21] and trying to fix or annotate all races when debugging [1]. Our parallel happens-before detector preserves this detection guarantee.

FastTrack uses *vector clocks* [14, 23] to precisely model the happens-before relationship of an execution[1]. A vector clock (VC) is a vector of N logical clocks, each of which represents a logical time for one of the N threads in the program. VCs are updated by merge and increment operations. A thread *merges* VC_b into VC_a by setting each element of VC_a to the maximum of its current value and the value of the corresponding element in VC_b. A thread i *increments* a VC by incrementing element i of that VC.

We associate a VC with each thread and each synchronization object. The VC for a thread tracks its local logical time and shows which logical times in other threads *happen before* the current instant in the thread's execution. The VC for a synchronization object shows which logical times in each thread *happened before* all releases to the object. When a thread acquires a synchronization object, the synchronization object's VC is merged into the thread's VC. When a thread releases a synchronization object, the thread's VC is merged into the synchronization object's VC, then the thread's VC is incremented.

As each memory access is handled, the detector checks to see if the access races with any previous access. It is sufficient to check that: (1) on a read of the variable, the read must *happen after* the last write to the variable, and (2) on a write, the write must *happen after* the last write of the variable as well as the last read of the variable by each thread. To implement these checks, we maintain for each variable a read-VC and a write-VC, which store the local logical clocks for a subset of the last read and write to the variable by each thread. A thread performs a check by comparing the read-VC and/or write-VC to the thread's VC. If each element of the variable's VC is less than or equal to the corresponding element of the thread's VC, then the check succeeds. Otherwise, the detector declares a race on the variable. After checking for a race, the variable's VC(s) are updated. On a write, the thread resets the read-VC and write-VC by setting all elements to zero, then writes its own logical time to its element in the write-VC (preserving only the last write). On a read, the thread writes its logical time to its element in the read-VC (preserving the last write and all other reads). If the read *happens after* all previous reads, the thread may also overwrite all other elements in the read-VC with zero (preserving only the last write and read).

3.2 Epoch-sequential phase

The first task in designing a parallel happens-before race detector is to decide what part of the analysis state to predict in the epoch-sequential phase. Recall that predicting this part of the analysis state should be much cheaper than the full race predictor, yet the partial state being predicted should enable the epoch-parallel phase to do most of the work of race detection.

It is instructive to consider the two extreme approaches we could take, neither of which work well. If the epoch-sequential phase predicted *all* analysis state, it would be as slow as the full race detector and the predictions would come too late to start new epochs. At the other extreme, if the epoch-sequential phase predicted *no* analysis state, each epoch would start without knowing any vector clocks values. Since VCs always evolve by merging, incrementing, or setting individual elements, no concrete VC values could be generated by the epoch-parallel phase, and the entire task of race detection would need to be deferred until the commit phase (when the VC values at the start of the epoch have been determined).

In our design, the epoch-sequential phase predicts the VCs for threads and synchronization objects, but it does not predict the read-VCs or write-VCs for variables. We generate this state by instrumenting only the synchronization operations. This subset of the analysis meets the criteria stated in Section 2.2.1. It is self-contained, since the VCs for threads and synchronization objects depend only on other VCs for threads and synchronization objects (and not on VCs for variables). It is lightweight, since variable accesses comprise the vast majority of events that need to be monitored. It will usually match the analysis state generated by the epoch-parallel execution—online replay forces the epoch-parallel execution to follow the happens-before order logged during the epoch-sequential execution, so the prediction of this analysis state can only diverge on a data race (which are rare and will be detected).

3.3 Epoch-parallel phase

The epoch-parallel phase performs most of the work of race detection. All program events, synchronization operations and memory accesses, are instrumented and handled using the full FastTrack algorithm. It starts with the predicted VCs for threads and synchronization objects and continues to maintain these during the epoch.

The epoch-parallel phase does *not* know the initial state of each variable's read-VC and write-VC at the beginning of the epoch, since these are not predicted by the epoch-sequential phase. We treat the whole write-VC and each unknown element of the read-VC as a symbolic variable. Whenever the race detection algorithm

[1] FastTrack represents most VCs using a sparse array rather than a naïve array. Although our implementation follows this same approach, we present the algorithm here without distinguishing whether a VC is represented sparsely or naïvely.

needs to evaluate an expression containing symbolic elements from read-VCs or write-VCs, it logs a symbolic form of the expression and defers its evaluation until the commit phase (when the initial read-VCs and write-VCs are known). This can occur when the detector compares a thread VC with a read-VC and/or write-VC in order to check for races.

When an epoch begins, the write-VCs and all elements of the read-VCs are symbolic, and all checks must be deferred. When a thread reads a variable and updates the thread's element in the read-VC, that index takes on a concrete value. Once all elements of a read-VC become concrete, the VC as a whole becomes concrete, and all comparisons against it can be evaluated immediately rather than being deferred. When a thread writes a variable, the entire read-VC and write-VC are given concrete values. Further comparison of that variable in the epoch can be evaluated immediately.

The system described up to this point defers an excessive amount of checks to be handled in the commit phase. We can use transitive reduction [28, 49] as an optimization to reduce the number of checks that must be deferred. Once a check has verified that an access b *happens after* a prior access a, we can henceforth assume that the check succeeded, even if that check is deferred because it involves a symbolic value. (If a deferred check later fails, it would reveal the first race on the variable, and finding subsequent races on that variable is not required.) Any subsequent access c that *happens after* b can therefore be assumed to *happen after* a. This reasoning often allows checks to be evaluated immediately, even when the read-VC and write-VC contain symbolic values. We use transitive reduction in two circumstances.

First, we do not need to log a check for a read that *happens after* a prior read in that epoch; we need only log checks for concurrent reads. Consider a read r_2 that *happens after* read r_1 in the same epoch. r_2 can race with the last write of a previous epoch only if r_1 also raced, so r_2 needs only to be checked against writes in the current epoch, and these checks can be evaluated immediately. When a read is handled, the detector compares the concrete portions of the read-VC to the thread's VC to decide if the read is concurrent with other reads in the epoch.

Second, consider a thread i that synchronizes (either directly or transitively) with another thread j during the epoch, i.e., thread i's VC happens after some event by thread j in this epoch. By transitivity, accesses by thread i after this point *happen after* all accesses in prior epochs by thread j. If thread i synchronizes with *all* threads during the epoch, subsequent accesses by thread i are guaranteed to happen after all accesses in prior epochs by all threads. In this situation, the detector need only check against races in the current epoch, and these checks can be evaluated immediately by examining the concrete values in the read-VCs and write-VCs.

As a result of these optimizations, we can determine a maximum bound on the number of checks per variables that can be deferred. In a process with N threads, there can be at most N concurrent reads, and there can be only a single first write. Hence, at most $N + 1$ checks can be deferred per variable. In particular, when all reads to a variable are totally ordered (e.g., data is thread-local or is protected by a lock), only 2 checks are deferred.

3.4 Commit phase

The commit phase performs the operations that were deferred during the epoch-parallel phase. Commit phases occur in program order, so the commit phase for one epoch has access to the concrete values of the final, complete analysis state for the prior epoch. In particular, it has access to concrete read-VC and write-VC values for the beginning of its epoch. It uses these concrete values to evaluate the symbolic expressions logged during the epoch-parallel phase and any symbolic values in the analysis state.

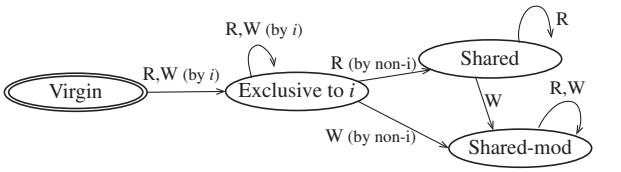


Figure 2. Eraser sharing state machine.

4. Parallel lockset

Another common approach to race detection is one based on *locksets* [39]. Lockset-based race detectors find potential races by looking for violations of a particular locking discipline. For example, a race may be declared for variables whose accesses are not consistently protected by at least one lock. Most lockset-based predictors are neither sound nor precise—they may miss real races and report false ones. Lockset detectors are nonetheless useful because they may find potential races that did not appear in the monitored execution.

We use the Eraser algorithm for lockset-based race detection [39]. Eraser tracks the *sharing state* of each variable according to the state machine shown in Figure 2. When a new variable is allocated, it starts in the *virgin* state. When the variable is first accessed, it transitions to the *exclusive* state and remains there until a different thread accesses it. If a different thread reads the variable, it transitions to the *shared* state; if a different thread writes the variable, it transitions to the *shared-modified* state.

The sharing state of a variable determines how the variable's lockset is updated. In the virgin state, the lockset is initialized to the set containing all locks in the program; it remains at this initial value while in the exclusive state. When the variable enters or is accessed in the shared or shared-modified states, its lockset is intersected with the locks currently held by the accessing thread (which we call the *thread lockset*). A race is declared if a variable's lockset is empty while the variable is in the shared-modified state. Eraser allows unlocked accesses to variables while they are being initialized by a single thread and to variables that are only read.

As described in Section 2.2, we split the task of race detection into three phases: epoch-sequential, epoch-parallel, and commit.

Overview: We instrument only synchronization and memory allocation in the epoch-sequential execution to predict each thread's lockset. The epoch-parallel execution adds memory access instrumentation, allowing it to track a superset of the true lockset for each variable. The commit phase merges a variable's lockset with that of previous epochs to reconstruct the actual lockset.

4.1 Epoch-sequential phase

The epoch-sequential phase predicts the thread locksets, but it does not predict the lockset or sharing state of each variable. To generate these predictions, synchronization and memory allocation operations are instrumented. This predicted subset of analysis state meets the criteria stated in Section 2.2.1. It is self-contained since thread locksets do not depend on variable states or variable locksets. It is lightweight since lock acquisitions and releases are much less frequent than variable accesses. It will usually match the analysis state generated by the epoch-parallel execution since the two executions can only diverge due to a data race. Because Eraser may miss data races, we verify that the predicted analysis state matches the analysis state generated by the epoch-parallel phase by comparing their values at the end of each epoch. In contrast, the parallel happens-before detector does not compare values because it identifies all data races.

	Initial sharing state {ex_sh, owner, modified}	
	{EX, NONE or i, *}	{SH, *, m_{init}} or {EX, $\neq$ NONE $\wedge \neq i$, *}
after access 1 (by thread i)	{EX, i, FALSE} $lockset = ALL$	{SH, *, $m_{\text{init}} \vee wr_1$} $lockset = LS_{\text{init}} \cap TLS_1$
after access 2 (by thread i)	{EX, i, FALSE} $lockset = ALL$	{SH, *, $m_{\text{init}} \vee wr_1 \vee wr_2$} $lockset = LS_{\text{init}} \cap TLS_1 \cap TLS_2$
after access 3 (by thread j)	{SH, *, wr_3} $lockset = TLS_3$	{SH, *, $m_{\text{init}} \vee wr_1 \vee wr_2 \vee wr_3$} $lockset = LS_{\text{init}} \cap TLS_1 \cap TLS_2 \cap TLS_3$
after access 4 (by thread k)	{SH, *, $wr_3 \vee wr_4$} $lockset = TLS_3 \cap TLS_4$	{SH, *, $m_{\text{init}} \vee wr_1 \vee wr_2 \vee wr_3 \vee wr_4$} $lockset = LS_{\text{init}} \cap TLS_1 \cap TLS_2 \cap TLS_3 \cap TLS_4$

Table 1. The two versions of analysis state maintained by epoch-parallel phase. The first version of analysis state assumes the initial sharing state is EX, and is owned either by NONE or by thread i (the thread that first accesses the variable in the epoch). The second version of analysis state covers all other conditions. wr_a is true iff access a is a write. TLS_a denotes the set of locks held by the accessing thread at the time of access a. LS_{init} and m_{init} denote the initial value of the variable's lockset and modification status. * means the value is ignored.

4.2 Epoch-parallel phase

The epoch-parallel phase performs most of the work of race detection. Synchronization, memory allocation, and memory access operations are all instrumented in this phase. An epoch starts with the predicted set of locks held by each thread and continues to maintain these during the epoch. However, the epoch-parallel phase does *not* know the initial value of each variable's lockset or sharing state, since these are not predicted by the epoch-sequential phase.

Not knowing the variables' initial sharing states has the potential to drastically slow down the epoch-parallel phase. The updates to a variable's lockset depends on the sharing state, so not knowing the sharing state forces the race detector to maintain multiple versions of each variable's lockset. For an application with N threads, there are $N + 3$ states in the sharing state machine: one exclusive state per thread, virgin, shared, and shared-modified. In the worst case, each state would cause the race detector to maintain the variable locksets differently, and this would require the race detector to perform $(N+3)$ times as many lockset manipulations as a conventional implementation.

Fortunately, by encoding and *factoring* the sharing state carefully, we eliminate most of this extra overhead. Instead of $N + 3$ versions of the sharing state and lockset, we maintain only two versions, and we need to manipulate only one version per variable access.

We encode the sharing state of a variable as a 3-tuple: {ex_sh, owner, modified}. The ex_sh field denotes whether the variable is exclusive to a single thread (EX) or shared by multiple threads (SH). If ex_sh is EX, then the owner field denotes which thread has accessed the variable, and modified is FALSE (the virgin state is handled as a special case of EX in which owner is NONE). If ex_sh is SH, owner is ignored, and the modified field denotes whether or not the variable has been modified while shared.

Table 1 shows how we maintain the sharing state and lockset for a variable, despite not knowing the initial sharing state or lockset. We maintain two versions of analysis state, which correspond to two possible categories of initial values for the sharing state. Conceptually, we maintain both versions of the sharing state and choose the correct one when the true sharing state at the beginning of the epoch is committed by the prior epoch. However, as we will show, our implementation needs only to update a single lockset and modified bit per access.

The first version of analysis state assumes the initial sharing state is EX, and owner is either NONE or the thread that first accesses the variable in the epoch. We call this *version-EX* because the first access in the epoch results in ex_sh=EX. The second version of analysis state is used for all other situations, i.e., it assumes the initial sharing state is SH, or it is EX and owner is a thread other than the one that first accesses the variable in the epoch. We call this *version-SH* because the first access in the epoch results in ex_sh=SH.

Consider the first access (in the epoch) to the variable by some thread i and all subsequent accesses by thread i that occur before some other thread accesses the variable. This sequence of accesses has different effects on the two versions of sharing state. For version-EX, this sequence of accesses leaves ex_sh=EX, owner=i, modified=FALSE, and lockset=ALL. For version-SH, this sequence of accesses leaves ex_sh=SH; the modified variable starts accumulating disjunctions of whether the access is a write or read; and the variable's lockset starts accumulating intersections of the accessing thread's lockset. Since disjunction and intersection are associative, we maintain these as a function of a symbolic value (m_{init} or LS_{init}) and a concrete value (e.g., in Table 1 after access 2, the initial values are m_{init} and LS_{init}, and the concrete values are $(wr_1 \vee wr_2)$ and $(TLS_1 \cap TLS_2)$). Note that, during this sequence of accesses, the epoch-parallel phase needs only to maintain the modified bit and the lockset for version-SH, since the modified bit and lockset for version-EX are not changing.

If a second thread accesses the variable in the epoch, we continue to maintain version-EX and version-SH. Since two threads have accessed the variable, ex_sh=SH in both versions. However, the value of modified and lockset continue to differ between the versions: in version-EX, these values are independent of the first sequence of accesses; in version-SH, these values depend on the first sequence of accesses. We minimize the number of operations needed to maintain modified and lockset for the two versions by factoring them into the disjunction (for modified) or intersection (for the lockset) of up to three expressions: the value at the beginning of the epoch (a symbolic value), a disjunction or intersection from the first thread's sequence of accesses (e.g., $TLS_1 \cap TLS_2$ in Table 1) and a disjunction or intersection from the accesses after the second thread accesses the variable (e.g., $TLS_3 \cap TLS_4$ in Table 1). With this factorization, we only need to manipulate a single instance of modified and lockset for each variable access.

4.3 Commit phase

The commit phase uses the final values for the lockset and sharing state from the prior epoch to generate the final values for the current epoch. It chooses between version-EX and version-SH based on the final values of ex_sh and owner from the prior epoch and the thread that first accessed the variable in the current epoch. If version-SH is chosen, it computes the final version of modified and lockset by merging in (via disjunction or intersection) the final value of modified or lockset from the prior epoch. After computing the final values for the analysis state, the commit phase detects if a race occurred on any variable accessed during this epoch by checking if any variable with ex_sh=SH has an empty lockset.

5. Implementation

Our execution system is based on the same infrastructure used in DoublePlay [44]. DoublePlay runs a 32-bit PAE Linux 2.6 kernel that has been heavily modified to provide uniparallelism. It uses online replay [19] to synchronize the epoch-parallel execution with the epoch-sequential execution, and it executes each epoch of the epoch-parallel execution on a single core. It also uses a custom version of glibc 2.5.1 to provide user-space logging for deterministic replay.

Race detectors need to analyze synchronization operations, memory management calls, and memory accesses. We intercept synchronization operations and memory management calls via dynamic library interposition. In particular, intercepting `malloc` calls allows lockset detectors to reinitialize analysis state on a new allocation. We intercept memory accesses by instrumenting loads and stores via a custom LLVM compiler pass. At default optimization levels, LLVM omits explicit loads and stores to variables whose references are proven not to escape the function. Consequently, we do not analyze accesses to these variables. For type-safe programs, these variables are guaranteed to be private to the thread. Since memory errors (e.g., buffer overflows) lead to undefined behavior and our race detector (like most such tools) runs in the address space of the instrumented process, such errors should be detected and fixed by other tools prior to running our detector.

For both happens-before and lockset race detectors, only the epoch-parallel execution needs to monitor loads and stores. For simplicity, we run the same program executable in both the epoch-sequential and the epoch-parallel execution, and the instrumentation checks a global variable to determine whether or not to analyze each instruction. The happens-before detector inlines this check within program instructions. The lockset detector (which compares the two executions' memory images) performs this check in its own function and additionally zeroes each new stack frame when it is allocated. A faster, though more complicated, strategy would be to run different program executables in the two executions [42]: the epoch-sequential execution would use a version without load/store instrumentation; the epoch-parallel execution would use a version with load/store instrumentation. We leave this for future work.

External libraries must either be compiled with instrumentation or have their functions annotated. We implement annotations for `glibc` through wrapper functions that explicitly invoke the handlers for analyzing loads and stores. For all other libraries, we compile a version with instrumentation.

Because race detectors analyze multithreaded programs, their analysis code must carefully handle the shared analysis state. Our analysis code uses spinlocks to provide atomicity at low latency in the common case. Analysis variables on a single cache line are guarded by the same lock to reduce inter-cache traffic. We provide a pool of spinlocks and hash the memory address being analyzed to select which spinlock to use. The pool contains 1024 spinlocks—large enough to avoid lock contention. These spinlocks are used by the conventional race detectors and the epoch-sequential phase of our race detector. The epoch-parallel phase instead ensures atomicity of the analysis code by running on a single core (Section 2.1).

For both happens-before and lockset detectors, all program memory is shadowed by analysis state, which is organized by a single-level page table with 4 MB pages. The baseline space overhead for the conventional and parallel happens-before detectors is twice the size of the program state; the baseline space overhead for the conventional and parallel lockset detectors is the size of the program state. Both parallel detectors use copy-on-write to minimize additional space overhead for the analysis state used by concurrent epochs; the amount of this additional space overhead is proportional to the number of epochs executing in parallel and the working set of each epoch.

The happens-before detectors find races with byte granularity. To minimize the analysis state and reduce the overhead of race detection, we compress the analysis state using a design similar to Valgrind's Memcheck [27]. We initially shadow entire four-byte words as single units. The first time an individual byte is accessed, we duplicate the word's analysis state and start shadowing each byte separately. The lockset detectors shadow and detect races with (fixed) word granularity without any compression.

6. Evaluation

This section evaluates experimentally the performance and scalability of our parallel happens-before and lockset race detectors. We compare the performance of our parallel race detectors with conventional happens-before and lockset detectors. The conventional detectors use the same race detection algorithms as the parallel detectors, which are based on FastTrack [15] and Eraser [39]. They use the same analysis infrastructure (LLVM) as the parallel detectors but run as normal multithreaded processes rather than using the uniparallel infrastructure. The main body of the analysis code is the same for conventional and parallel detectors, except that the parallel detectors do extra work to split the work across the epoch-sequential, epoch-parallel, and commit phases, while the conventional detectors acquire and release locks to guarantee atomicity for the analysis code. We compare the performance of all race detectors (both parallel and conventional) against uninstrumented versions of each benchmark that perform no race detection.

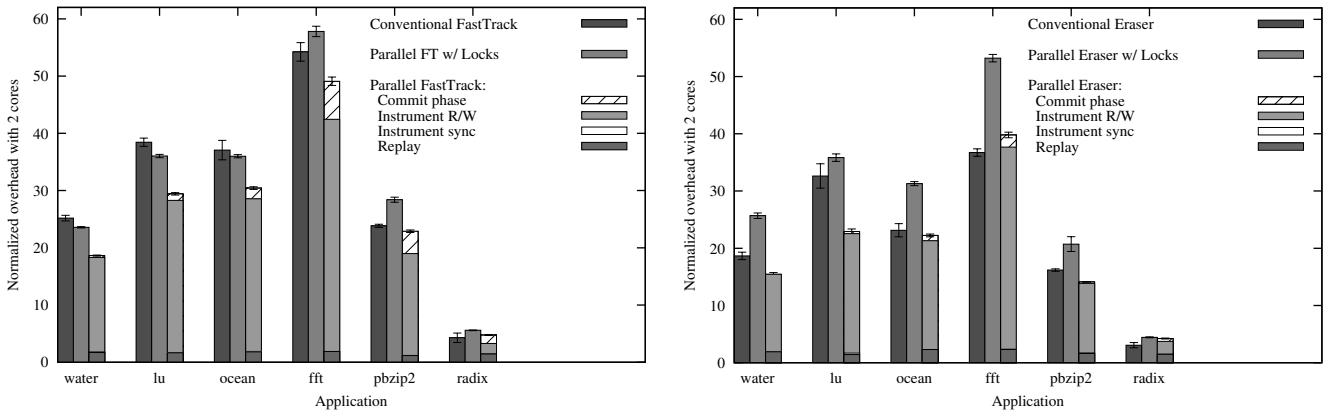
All experiments were performed on an 8-core workstation with 4 dual-core 2.4 GHz Intel Xeon CPUs (12 MB cache) and 6 GB RAM. We evaluate performance on five applications from the modified SPLASH-2 benchmark suite [48] (water-n^2, lu, ocean-contig, fft, radix) and one parallel application (pbzip2). The SPLASH-2 applications are CPU and memory intensive and contain little I/O. Pbzip2 is used to compress an 8 MB file, with multiple threads compressing different blocks of the file in parallel. We prefetch this file in advance to keep the application from becoming I/O-bound. We configure each application to use two worker threads; this allows us to evaluate scalability up to a 4:1 ratio of cores to threads; experiments with four worker threads had similar results up to the maximum ratio (2:1) on our platform. We exclude many of the other SPLASH-2 benchmarks because they contain frequent data races. The performance of a race detector is most important for programs that are mostly race-free and that run for an extended amount of time before encountering a data race (frequent races can be detected quickly even with a slow detector). When a race is detected, our detector records the event, but does not otherwise alter the execution of the target program.

We adjust the workload sizes so that each SPLASH-2 program executes in about 2 minutes on a single core when no race detection is used; pbzip2 take about 1 minute to run. For some applications, we scale the execution by looping through program's main computation multiple times. We also modify some applications to make periodic system calls, which allows our infrastructure to break long-running epochs.[2] All measurements are the mean of at least 4 samples of a benchmark's overall execute time. All graphs show 90% confidence intervals.

6.1 Overhead

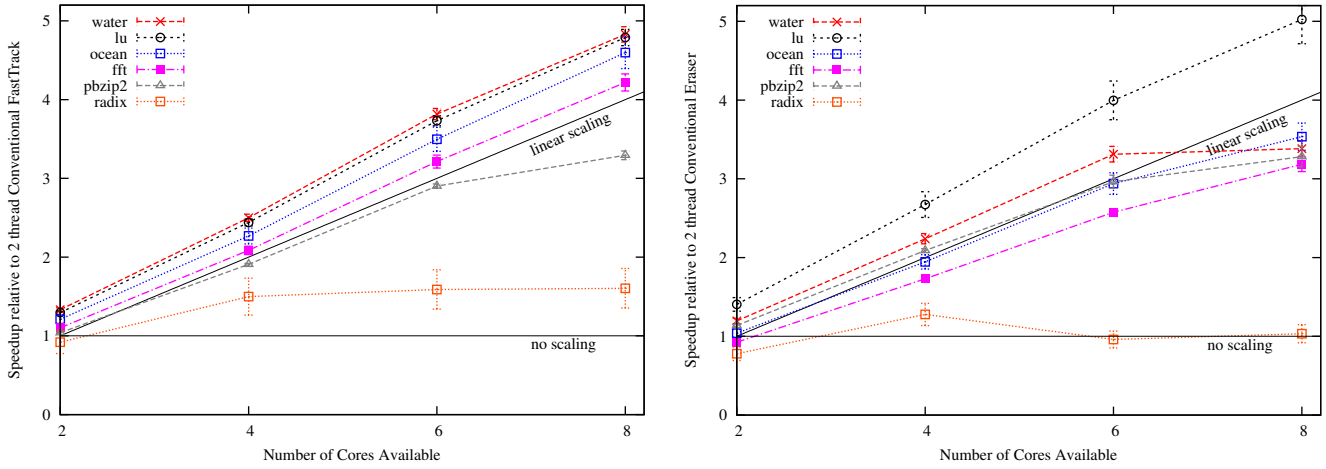
We start by evaluating the performance of the conventional and parallel race detectors on a system with one core per worker thread (i.e., two cores total). Figure 3 shows the overhead of each detector, normalized to the execution time of the original application run without any race detector. For each application, the left bar shows

[2] The deterministic replay system we used ends epochs only at synchronous preemption points.



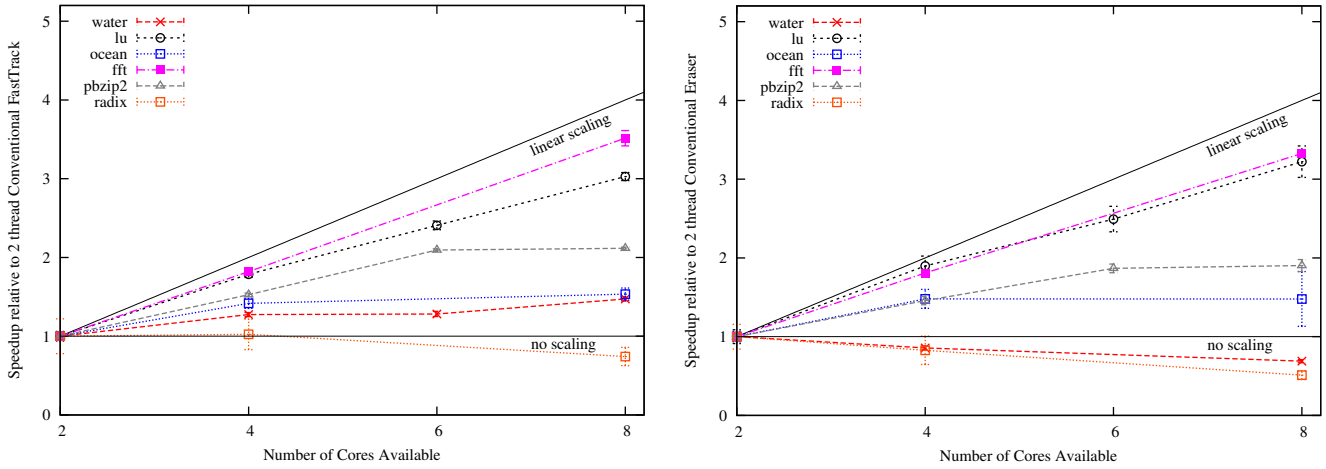
(a) Happens-before (b) Lockset

Figure 3. Overhead of race detection. Overhead values are normalized to the application's running time without race detection. For each application, the left bar shows the overhead of the conventional race detector; the right bar shows the overhead of the parallel race detector; and the middle bar shows the overhead the parallel race detector would have if it did not benefit from eliding analysis locks. The workloads shown use two worker threads and are run on two cores for both detectors.



(a) Happens-before (b) Lockset

Figure 4. Scaling race detection via uniparallelism. Speedup is shown relative to the performance of the conventional race detector when run on two cores.



(a) Happens-before (b) Lockset

Figure 5. Scaling conventional race detection by increasing the number of worker threads in the application. Speedup is shown relative to the performance of a 2-thread application using the conventional race detector. In all cases, the number of worker threads in the application is equal to the number of cores used.

the overhead of the conventional race detector, and the right bar shows the overhead of the parallel race detector. The middle bar shows the overhead the parallel race detector would have if it did not benefit from eliding analysis locks.

Overhead varies widely between applications, depending on how frequently they access memory. Radix stands out as having the lowest overhead for race detection, due to radix's relatively low ratio of non-stack memory accesses to computation. As we will see later, radix's low overhead leaves little room for improvement via parallelization.

The middle bars in Figure 3 shows the overhead the parallel race detector would have if it needed to acquire and release analysis locks. Such a detector is slower than the conventional detector, since it takes extra work to parallelize an algorithm. For the lockset detector, lockset factorization greatly reduces the amount of extra work that needs to be done by the parallel detector.

The right bars in Figure 3 show that uniparallelism's ability to elide analysis locks usually gains back the extra overhead caused by parallelizing the algorithm. In fact, eliding analysis locks usually allows the parallel detectors to outperform the conventional ones, even when run on the same number of cores.

Overall, when both are run on the same number of cores, the parallel happens-before detector is 25% faster to 9% slower than the conventional detector (median 13% faster), and the lockset-based detector is 28% faster to 29% slower than the conventional detector (median 8% faster).

The rightmost bar in Figure 3 breaks the overhead for the parallel detectors into several components. *Replay* shows the time added to replay the application via uniparallelism without performing analysis on the replaying run. *Instrument sync* shows the time added to analyze synchronization operations in the epoch-sequential phase. *Instrument read/write* shows the time added to analyze memory accesses. *Commit phase* shows the time added to maintain and process the log of work that is deferred during the epoch-parallel phase.

We measured the overhead added by each component by incrementally enabling components of the instrumentation. For example, we measured overall running time with replay enabled but no instrumentation or commit; then we measured the running time with replay and instrumented synchronization calls but no read/write instrumentation or commit; etc.

As expected, most overhead comes from analyzing memory accesses. This overhead parallelizes well because the work to analyze memory accesses can be performed in the epoch-parallel phase. Commit phases for multiple epochs execute sequentially in our system, so they are a potential scalability bottleneck; however, Figure 3 shows that the commit phase is usually a small fraction of the overall overhead. For the happens-before detector, transitive reduction reduces the number of checks that are deferred to the commit phase; for the lockset detector, the commit phase has little work to do for each variable that was accessed in the epoch. Analyzing synchronization operations is done in the epoch-sequential phase in both detectors and could therefore be a scalability bottleneck, but Figure 3 shows that it adds a negligible amount to runtime.

6.2 Scalability

Our primary goal is to parallelize race detection across the available cores in the system. For benchmarks with two worker threads, we conduct experiments on 2, 4, 6, and 8 cores.

Figure 4 shows the speedup of our race detectors as we scale between 2 and 8 cores. Each point shows the speedup relative to the running time of the conventional race detector on 2 cores. The happens-before race detector parallelizes well for all applications except radix: with 8 cores, the median speedup over a 2-core conventional race detector is 4.4× (range is 1.6-4.8×). Parallel lockset also scales well for all applications except radix, but begins to reach scalability limits at 6 cores: with 6 cores, the median speedup over a 2-core conventional race detector is 3.0× (range is 1.0-4.0×). The lockset-based detector scales less well than the happens-before detector because the lockset detector adds less amount of overhead, so there is less room for improvement when scaling to more cores. The limiting factor to scaling is the the infrastructure we use to instrument the epoch-sequential execution. Because we run the same executable in both epoch-sequential and epoch-parallel executions, the epoch-sequential execution must perform some work on each memory access (viz., it must detect that it is running in the epoch-sequential execution and decide not to analyze the memory access). This per-access instrumentation causes the epoch-sequential execution to take about 3-5× as long as the original application. To scale the overhead lower than this, the epoch-sequential execution could use a different executable that eliminated this check [42].

For comparison, we show the speedup for two hypothetical parallel versions of the conventional detectors. First, the *linear scaling* lines in Figures 4 show the ideal speedup of the conventional detectors on 2 cores, assuming it could scale proportionally with available cores.

Second, we evaluate another strategy for parallelizing conventional detectors, which is to scale the application itself to use more cores. Figure 5 shows the speedup of this strategy, relative to the running time of the 2-thread application with a conventional race detector on 2 cores. Figure 5 shows that this strategy does not scale as well as our parallel detectors, in part because the applications themselves do not scale perfectly. Besides poorer scalability, there are two other downsides to this strategy. First, it works only for applications that can be easily configured to use more cores. Second, this strategy works only if the scaled application exhibits the same bugs as the original application. In contrast, parallelizing the race detectors via uniparallelism can achieve speedups on the original application, and it does not require the application itself to scale.

7. Related Work

Many researchers have studied how to detect data races dynamically via software techniques [2, 8, 9, 12, 15, 16, 32, 34, 35, 39, 50] and hardware techniques [11, 24, 30, 33, 51]. Despite much progress, dynamic data race detection remains expensive on commodity hardware, especially when analyzing unmanaged code.

Epoch parallelism was first used by Zilles and Sohi [52], who called it master/slave speculative parallelization. In their work, a fast, approximate version of a program to speed up a slow, correct version of the program. Others have since used epoch parallelism to parallelize some program analyses, such as data cache simulation [47], dynamic information flow tracking [29, 37], memory safety checks [17, 42], and program assertions [42]. Like us, Ruwase, et al. track symbolic values during an epoch and resolve them with concrete values when earlier epochs finish [37].

Uniparallelism, which is a variant of epoch parallelism, was first used by Veeraraghavan, et al. to implement a fast, deterministic replay system for multithreaded programs [44]. They used uniparallelism to minimize thread switches and thereby reduce the amount of nondeterminism that need to be logged. We use uniparallelism to eliminate thread switches while running analysis code, which reduces overhead by eliminating lock acquisitions and releases.

Veeraraghavan, et al. also used uniparallelism to design an *outcome-based* data race detector [43]. This algorithm detects races by running multiple replicas with different schedules and comparing their results [30]. Their outcome-based race detectors has low overhead but allows false negatives because it fails to detect races that do not affect the state at the end of the epoch. In contrast, we use uniparallelism to accelerate two classic types of data race detectors, namely happens-before and lockset analysis.

To our knowledge, ours is the first work that parallelizes conventional data race detectors on commodity hardware in a scalable way. Other researchers have split race detection across two processors by decoupling the monitoring of memory accesses from the race analysis of those accesses [7, 36], but this approach requires hardware support to reduce the overhead of instrumenting memory accesses and shipping logs to other processors. Uniparallelism allows us to parallelize both the instrumentation and analysis of memory accesses, and it reduces interprocessor communication by allowing these two tasks to run on a single processor for each epoch.

One benefit we gain from uniparallelism is its ability to provide mutual exclusion around analysis code without the overhead of acquiring and releasing locks. There are several other ways to provide mutual exclusion when analyzing multithreaded programs. The most common approach is to use coarse or fine-grained locking [12, 26, 38, 40], but this leads to significant overhead for race detectors because they need to acquire and release locks frequently and because they write analysis data even when reading application variables [45]. Using lock-free data structures can reduce this overhead, but these structures are more complicated to use than locks and cannot easily be applied in every situation (e.g., RaceTrack [50] uses lock-free lookups into a shared table, but it must lock when adding entries). Other researchers have proposed using hardware support for transactional memory to achieve the performance of fine-grained analysis locks [10]. Through uniparallelism, our system achieves the simplicity of a single lock, still allows analysis code to run in parallel, and eliminates the overhead of acquiring and releasing any locks. Our results show that eliding locks in this manner can significantly speed up race detection; this same approach could be used to speed up other high-frequency analyses.

8. Conclusions

This paper has shown how to speed up race detection via uniparallelism. Uniparallelism executes epochs across cores to increase parallelism, and it executes each epoch on a single core to reduce analysis locking overhead. We divide race detectors into three phases to enable them to run in a uniparallel environment: the epoch-sequential phase predicts a subset of the analysis state, the epoch-parallel phase carries out the bulk of the analysis, and the commit phase resolves symbolic expressions logged during the epoch-parallel phase.

We demonstrated our strategy by parallelizing a happens-before race detector and a lockset-based race detector. Each detector required unique optimizations to work well across the three phases: the happens-before detector used transitive reduction to reduce the work that must be deferred to the commit phase, and the lockset-based detector used lockset factorization to reduce the work needed to maintain multiple lockset versions during the epoch-parallel phase.

Our results showed that race detection can be parallelized effectively via uniparallelism. With 4× the number of cores as the original application, our strategy sped up the median execution time by 4.4× for a happens-before detector and 3.3× for a lockset race detector. Even on the same number of cores as the conventional detectors, the ability for uniparallelism to elide analysis locks allowed it to reduce the median overhead by 13% for a happens-before detector and 8% for a lockset detector.

We believe that many of the lessons learned in parallelizing data race detection can be applied to other heavyweight analyses of multithreaded programs. Analyses that can be restructured into the above three phases can benefit from uniparallelism, as long as most work can be done in the epoch-parallel phase. For any analysis, a key aspect of this restructuring is selecting the subset of the analysis to predict in the epoch-sequential phase. In addition, analyses that frequently acquire and release locks will benefit from the ability of uniparallelism to elide those locks.

Acknowledgments

We would like to thank all the anonymous reviewers for their valuable feedback and suggestions. This work has been supported by the National Science Foundation under award CNS-0905149 and by Intel Corp.. Satish Narayanasamy is supported by NSF CAREER award CNS-1149773. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Facebook, the University of Michigan, or the U.S. government.

References

[1] S. Adve. Data races are evil with no exceptions. *Communications of the ACM*, 53(11):84, November 2010.

[2] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *Proc. 1991 International Symposium on Computer Architecture*, pages 234–243.

[3] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen. A theory of data race detection. In *2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 69–78.

[4] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation*, pages 68–78.

[5] H.-J. Boehm and S. V. Adve. You Don't Know Jack About Shared Variables or Memory Models. *Communications of the ACM*, 55(2): 48–54, February 2012.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proc. 2010 ACM Conference on Programming Language Design and Implementation*, pages 255–268.

[7] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-Based Architectures for General-Purpose Monitoring of Deployed Code. In *2006 Workshop on Architectural and System Support for Improving Software Dependability*.

[8] 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 *Proc. 2002 ACM Conference on Programming Language Design and Implementation*, pages 258–269.

[9] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, October 1991.

[10] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-Safe Dynamic Binary Translation Using Transactional Memory. In *Proc. 2008 Symposium on High Performance Computer Architecture*, pages 279–289.

[11] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *Proc. 2012 International Symposium on Computer Architecture*, pages 201–212.

[12] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 245–255.

[13] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proc. 2010 Symposium on Operating Systems Design and Implementation*, pages 151–162.

[14] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10 (1):56–66, February 1988.

[15] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proc. 2009 ACM Conference on Programming Language Design and Implementation*, pages 121–133.

[16] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *Journal of Parallel and Distributed Computing*, 59(2):180–203, November 1999.

[17] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A software system for speculative program optimization. In *Proc. 2009 International Symposium on Code Generation and Optimization*, pages 157–168.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[19] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocesor replay via speculation and external determinism. In *Proc. 2010 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–90.

[20] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[21] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proc. 2010 International Symposium on Computer Architecture*, pages 210–221.

[22] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proc. 2009 ACM Conference on Programming Language Design and Implementation*, pages 134–143.

[23] F. Mattern. Virtual time and global states of distributed systems. In *Proc. 1988 International Workshop on Parallel and Distributed Algorithms*.

[24] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *Proc. 2009 International Symposium on Computer Architecture*, pages 337–348.

[25] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 22–31.

[26] N. Nethercote and J. Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 89–100.

[27] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proc. 2007 ACM Conference on Virtual Execution Environments*, pages 65–74.

[28] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *1993 ACM/ONR Workshop on Parallel and Distributed Debugging*.

[29] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. 2008 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318.

[30] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight hardware support for data race detection using systematic testing of parallel programs. In *Proc. 2009 International Symposium on Microarchitecture*, pages 541–552.

[31] K. Poulsen. Tracking the blackout bug. Technical report, SecurityFocus, April 2004. http://www.securityfocus.com/news/8412.

[32] E. Pozniansky and A. Scheuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. 2003 ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–190.

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

[34] P. Ratasaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *Proc. 2009 ACM Symposium on Principles and Practice of Parallel Programming*, pages 173–184.

[35] M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17 (2):133–152, May 1999. ISSN 0734-2071.

[36] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 245–255.

[37] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proc. 2008 ACM Symposium on Parallelism in Algorithms and Architectures*, pages 35–45.

[38] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel Thread Checker race detector. In *Proc. 2006 Workshop on Architectural and System Support for Improving Software Dependability*, pages 34–41.

[39] 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, November 1997.

[40] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proc. 2009 Workshop on Binary Instrumentation and Applications*, pages 62–71.

[41] J. Sevcik and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proc. 2008 European conference on Object-Oriented Programming*, pages 27–51.

[42] M. Süßkraut, T. Knauth, S. Weigert, U. Schiffel, M. Meinhold, and C. Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proc. 2010 International Symposium on Code Generation and Optimization*, pages 131–140.

[43] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proc. 2011 ACM Symposium on Operating Systems Principles*, pages 369–384.

[44] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proc. 2011 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26.

[45] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *Proc. 2010 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–284.

[46] C. von Praun and T. R. Gross. Object race detection. In *Proc. 2001 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82.

[47] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. 2007 International Symposium on Code Generation and Optimization*, pages 209–220.

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

[49] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (RTR) for longer memory race recording. In *Proc. 2006 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–60.

[50] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proc. 2005 ACM Symposium on Operating Systems Principles*, pages 221–234.

[51] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *Proc. 2007 Symposium on High Performance Computer Architecture*, pages 121–132.

[52] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proc. 2002 International Symposium on Microarchitecture*, pages 85–96.