# Chimera: Hybrid Program Analysis for Determinism

Dongyoon Lee Peter M. Chen Jason Flinn Satish Narayanasamy
University of Michigan
{dongyoon,pmchen,jflinn,nsatish}@umich.edu

## Abstract

Chimera[1] uses a new hybrid program analysis to provide deterministic replay for commodity multiprocessor systems. Chimera leverages the insight that it is easy to provide deterministic multiprocessor replay for data-race-free programs (one can just record non-deterministic inputs and the order of synchronization operations), so if we can somehow transform an arbitrary program to be data-race-free, then we can provide deterministic replay cheaply for that program. To perform this transformation, Chimera uses a sound static data-race detector to find all potential data-races. It then instruments pairs of potentially racing instructions with a *weak-lock*, which provides sufficient guarantees to allow deterministic replay but does not guarantee mutual exclusion.

Unsurprisingly, a large fraction of data-races found by the static tool are false data-races, and instrumenting them each of them with a weak-lock results in prohibitively high overhead. Chimera drastically reduces this cost from 53x to 1.39x by increasing the granularity of weak-locks without significantly compromising on parallelism. This is achieved by employing a combination of profiling and symbolic analysis techniques that target the sources of imprecision in the static data-race detector. We find that performance overhead for deterministic recording is 2.4% on average for Apache and desktop applications and about 86% for scientific applications.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging; D.4.5 [*Operating Systems*]: Reliability

***General Terms*** Design, Performance, Reliability

***Keywords*** Determinism, Replay, Data-race detection, Static analysis, Profiling, Symbolic range analysis

## 1. Introduction

A shared-memory multithreaded program is not guaranteed to produce the same output across different executions even if the input is guaranteed to be the same. Lack of determinism significantly impairs a programmer's ability to reason about an execution and understand the root causes of program failure. This problem can be addressed by constructing multiprocessor systems that guarantee that every execution for a given input produces the same output [14]. Another approach is to record the non-deterministic thread interleavings and enable a programmer to deterministically replay and understand an execution [28]. In this paper, we focus on solving the latter replay problem for multithreaded programs, but the principles discussed here could be applied to build a deterministic multiprocessor system as well.

The ability to faithfully reproduce an execution has proven useful in many areas, including debugging [27, 47], fault tolerance [11], computer forensics [16], dynamic analysis [13, 36], and workload capture [34]. However, past solutions to deterministic replay for shared-memory multiprocessor systems have been unsatisfactory either due to performance costs [17, 29, 49], reliance on custom hardware [24, 32, 33, 54], or lack of sufficiently strong determinism guarantees [2, 38, 51, 56].

Uniprocessor replay is relatively easy. During a recording phase, non-deterministic events (e.g., interrupts and data read from input devices) are logged. If the program is multithreaded, then thread schedules (e.g., the instructions at which each thread is preempted) must also be logged and replayed [43]. Previous studies have shown that logging these non-deterministic events adds little overhead [55].

Efficient multiprocessor replay, however, remains an open problem. The fundamental challenge comes from recording the non-deterministic interleaving among threads, a property that is necessary to deterministically replay programs that contain data-races. Recording and replaying the frequent interactions among thread accesses to shared data can slow execution by an order of magnitude or more. However, if one could somehow statically guarantee that a program is data-race-free, then it is not necessary to record thread interactions — past research has shown that recording and replaying the happens-before order of synchronization operations is sufficient to ensure deterministic replay [40]. In most applications, synchronization operations are relatively infrequent compared to memory accesses, and therefore logging them is relatively cheap. Many uses of deterministic replay, including program debugging, reproducing errors encountered in the field in a test environment, replication for fault tolerance, and forensics following a computer intrusion, are especially useful for programs with bugs, so working only for bug-free programs (i.e., programs without data-races) is not a viable option.

Unfortunately, statically proving the absence of data-races in a program without rejecting data-race-free programs is hard. If there is a chance that a program contains a data-race, then one must record the order of potentially racing operations in order to guarantee that the recorded program can be replayed deterministically. One could discover such operations with a dynamic data-race detector. However, despite significant advances, dynamic data-race detection in software slows program execution by nearly 8x [20] for state-of-the-art detectors. Thus, logging the order of potentially

[1] Chimera is a mythological hybrid animal composed of parts of a lion, a goat and a snake.

racing instructions is no less of a problem than detecting a data race.

In this paper we discuss Chimera, a deterministic replay system that employs a new hybrid program analysis to handle programs with data races. Chimera combines static data race analysis with off-line profiling and targeted, dynamic checks to provide deterministic replay efficiently.

Chimera instruments a program to log all non-deterministic inputs (e.g., system call results), the thread schedule on each processor core, and the happens-before relationships due to synchronization operations. This information is sufficient to guarantee that the program can later be replayed deterministically, provided the program contains no data-races.

To provide replay for racy programs, Chimera uses a sound but imprecise static data-race detector (RELAY [50]) to find potential data-races. Every memory instruction that potentially races with another instruction is placed inside a code region protected by a *weak-lock*. Chimera records all happens-before relationships due to weak-locks in addition to the relationships due to the original program synchronization. Thus, Chimera guarantees deterministic replay for all programs.

We use weak-locks instead of traditional locks in order to be conservative and avoid introducing artificial deadlocks. A weak-lock is essentially a time-out lock, where mutual exclusion is compromised if the weak-lock is not acquired in reasonable amount of time. In the rare case when a weak-lock times out, Chimera deterministically preempts the thread that currently holds the weak-lock and forces it to yield the weak-lock to the thread that timed out on the weak-lock; the original holder of the weak-lock must reacquire the weak-lock before resuming its execution. This approach splits the code region protected by the weak-lock into two regions across the preemption. Because this timeout mechanism enables Chimera to preserve the invariant that only one thread holds a given weak-lock at any given time, Chimera can support deterministic replay by reproducing the order of weak-locks at the same preemption point.

Unsurprisingly, we find that a sound data-race detector reports a large number of false data-races, and thus adding a weak-lock for every reported data-race results in prohibitively high overhead. Chimera employs two critical optimizations to drastically reduce this cost.

Both optimizations attempt to increase the granularity of a weak-lock, in terms of the size of the locked code region and the amount of data the lock protects. Coarser weak-locks reduce the cost of instrumentation but may serialize threads unnecessarily and compromise parallelism. Chimera's optimizations navigate this performance trade-off by targeting the main sources of imprecision in a static data-race detector [50].

The first optimization is based on the observation that a large fraction of false data-race reports are due to the inability of the static data-race detector to account for the happens-before relations due to synchronization operations other than locks. One example of this is that a number of data-races are reported between initialization code and the rest of the program because the static tool does not account for the happens-before relation due to fork-join synchronization. To address this problem, we profile the program offline over a variety of inputs. If the code regions containing potentially racing instructions are non-concurrent in all profile runs, Chimera increases the granularity of the weak-lock to protect the entire code region instead of just one instruction. Chimera currently treats functions as code regions. This optimization reduces the number of times weak-locks are acquired and released during a function's execution.

The second optimization pertains to the remaining set of false racy pairs that are part of function pairs that ran concurrently in at least one profile run. This optimization targets the inaccuracy caused by the conservative points-to analysis [3, 45] on which RELAY is based. Due to this analysis, RELAY overestimates the set of shared objects that could be accessed by a memory instruction and also underestimates the set of locks that could be acquired. We observe that while the numeric values for the address bounds of an object accessed by a memory instruction are generally hard to determine precisely during static analysis, one can often estimate reasonable bounds in the form of a symbolic expression [42].

Therefore, in our implementation, we compute symbolic address bounds of objects that can be accessed by a racing instruction within a loop. Using this information, we increase the granularity of the weak-lock to the entire loop containing the race, such that it protects the loop for the data variables specified by the loop's symbolic address bounds. This avoids the cost of instrumentation for every iteration of the loop.

Our evaluation shows that Chimera is more efficient than the state-of-the-art software solutions that guarantee multiprocessor replay [49]. We show that recording a set of server (e.g., Apache) and desktop (e.g., pbzip, aget) applications incurs only about 2.4% performance overhead, and recording a set of memory-intensive scientific applications (SPLASH [53] incurs about 86%. Replay overhead is also similar to that of recording. We find that our two optimizations play a significant role in bringing the average overhead from 53x (when all races are naively instrumented) to 1.39x.

Programs transformed by Chimera are data-race-free under the new set of synchronization operations. Though our immediate motive for this transformation is to provide deterministic record and replay, we envision that future work may be able to leverage the data-race-freedom provided by Chimera to provide stronger guarantees such as sequential consistency and deterministic execution [37], since these properties are much easier to guarantee in the absence of data-races.

The primary contributions of this paper are as follows:

- We discuss Chimera, a new deterministic replay system for commodity multiprocessors based on a static data-race detector.
- We discuss two optimizations that employ profiling and symbolic bounds analysis to drastically reduce the overhead of a naive method that instruments all false data-races.
- Our experimental study shows that the performance overhead is about 40% on average, which is less than the state-of-the-art software solutions for multiprocessor replay.

## 2. Design Overview

This section provides a design overview of the Chimera multiprocessor replay system.

### 2.1 Background

A program is said to be data-race-free if none of its executions exhibit a data-race. Two memory instructions are said to be *racy* if at least one of them is a write, and there is at least one execution where the two are executed in different threads and not ordered by any happens-before relation due to synchronization operations. For clarity, we define a few terms that we use in this paper. A race-pair is a pair of static memory instructions that are racy. The two functions (or loops) that contains the race-pair are referred to as a racy-function-pair (or a racy-loop-pair).

Chimera records non-deterministic input (e.g., interrupts and file reads) and happens-before relations due to synchronization accesses in a program. This is sufficient to later provide deterministic replay for data-race-free programs because all memory instructions are ordered by some happens-before relation [40]. However, it is

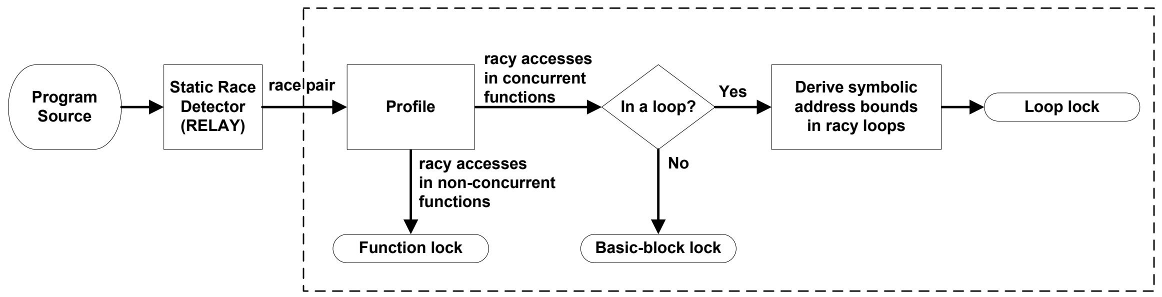

Program Source
Static Race Detector (RELAY)
race pair
Profile
racy accesses in concurrent functions
In a loop?
Yes
Derive symbolic address bounds in racy loops
Loop lock
racy accesses in non-concurrent functions
No
Function lock
Basic-block lock

**Figure 1.** Chimera Overview

insufficient to later provide deterministic replay for programs that contain data-races.

## 2.2 Design Overview

Figure 1 presents an overview of how Chimera transforms a potentially racy program to a data-race-free program by adding additional synchronization and runtime constraints. Chimera's transformation does not attempt to correct a given program, it simply makes it easier to deterministically record and replay the program's execution.

Chimera analyzes a given program using the RELAY [50] static data-race detector. RELAY is sound, except for two corner cases (assembly instructions and pointer arithmetic). However, the unsoundness is modularized and can be addressed using additional analysis [5, 52] (Section 3.2).

In the simplest implementation of Chimera, each race-pair is placed inside a code section protected by an unique weak-lock $w$. Recording and replaying the happens-before relation due to weak-locks enables Chimera to record and replay the order of all racy accesses and thus guarantee deterministic replay for all programs.

A sound static data-race detector is imprecise as it has to make very conservative assumptions. This results in a huge number of false data-races, and naively recording all those races results in high overhead. The insight of this paper is that by employing a combination of profiling, symbolic address bounds analysis and dynamic checks, the overhead is significantly reduced to the point where deterministic record and replay is viable even for production systems.

We discuss two specific optimizations. The first optimization is based on our observation that for many false race-pairs, the code regions containing them are almost never executed concurrently. One main cause for this imprecision is the static data-race detector's inability to account for non-mutex synchronization operations. Chimera learns which code regions are almost always non-concurrent by profiling executions with a set of representative inputs. It uses profile information to increase the granularity of weak-locks, both in terms of size of the code region and the amount of shared objects they protect, which reduces the number of weak-lock operations at runtime. Chimera's profiler treats every function as a code region, though other granularities could be considered. As shown in Figure 1, racy-pairs in non-concurrent functions are handled using weak-locks instrumented at the granularity of a function (referred to as *function-locks*).

Not all false data-races are part of non-concurrent code regions. Two code regions can overlap in time, but still may not exhibit a data race if they access different sets of shared objects. However, a static data-race detector may not always be able to prove that the set of shared objects accessed in concurrently executed code regions are disjoint due to imprecise pointer analysis. While it is hard to accurately compute the numeric values for address bounds statically for a code region, it is often possible to derive a symbolic expression for the upper and lower bounds of an object that will be accessed within a code region [42].

For data-races that are not found to be part of non-concurrent functions, Chimera checks if they are part of a loop. If a data-race is not part of any loop, then Chimera simply instruments a weak-lock at the granularity of a basic block (referred to as a *basic block lock*). In case the basic block has a function call, Chimera instruments a weak-lock at the granularity of an instruction (referred to as an *instruction lock*).

If a data-race is part of a loop, Chimera derives a symbolic address bound for the range of addresses that a racy instruction can access within the loop. A race-pair is then guarded by instrumenting a loop-lock. The loop-lock is also a weak-lock, but it protects a range of addresses, which are computed at runtime using the symbolic expression derived statically. If the symbolic bounds expression is too imprecise (e.g., one of the bounds is infinity), and if the loop body is reasonably large in size, then Chimera instruments at the granularity of a basic block. In this manner, Chimera avoids the risk of over-serializing the execution of loops.

## 2.3 Weak-Lock Design

Chimera ensures that the instrumented weak-locks do not introduce a deadlock. Chimera orders the set of weak-locks constructed for each granularity of a code region (basic block, loop, and function) and ensures that they are always acquired in the same order. When a program has nested code regions (e.g., a function calling a function, a loop calling a function, etc.), an outer region releases all its weak-locks before starting the inner region, and acquires the weak-locks back after exiting the inner region. The order in which weak-locks of different granularities are acquired is also consistent. Function-locks are always acquired before loop and basic-block locks. Loop-locks are always acquired before basic-block locks. Hence, there cannot be a deadlock between weak-locks.

Chimera avoids deadlocks that may happen when a weak-lock protected code region contains a programmer specified synchronization wait. The "weak" part of the weak-lock is meant for handling such deadlocks. If a weak-lock is stalled for more than a threshold period of time, the stalled weak-lock invokes a special system call to handle the potential deadlock. The system call handler identifies the thread that currently owns the stalled weak-lock by examining the log files used to record the order of weak-lock acquires and releases. The kernel preempts the current owner, and forces it to release and reacquire the weak-lock that timed-out. This allows the stalled thread to acquire the weak-lock and proceed with its execution.

Though the above mechanism may compromise the atomicity of a weak-lock protected code region, we always preserve the invariant that only one thread holds a given weak-lock at any given time. Thus, recording and replaying the exact order of forced weak-lock release and reacquire operations with respect to instrumented weak-lock operations is sufficient to guarantee deterministic replay.

This requires that Chimera record and replay the exact instance when a thread is preempted and forced to release its weak-locks. For this purpose, we plan to use a mechanism from the DoublePlay replay system [49] in which the kernel records the instruction pointer and the branch count (measured via hardware performance counters) at the point of preemption. We have not yet ported this implementation to the Chimera infrastructure as none of our benchmarks have exhibited a weak-lock timeout.

### 2.4 Discussion

Any data-race that exists in the original program can manifest in the transformed program. However, Chimera now records the order between the racing instructions. Increasing the granularity of weak-lock (e.g., to a basic-block) would make it less likely for instructions from two racy basic-blocks to interleave. If there is only one race between two racy basic-blocks, then all thread interleavings in the original program can manifest at approximately the same probability in the transformed program. However, if there is more than one race between two basic blocks, then Chimera's weak-locks will try to serialize them. While preventing fine-grained interleaving of smaller code regions may be beneficial for masking certain atomicity violations in production systems [31], a programmer trying to record and debug a test run might consider this to be a limitation of Chimera's optimizations.

## 3. Static Data-Race Detection

Chimera uses the RELAY [50] static data-race detector to identify potential data-races. In this section, we briefly summarize the RELAY detection algorithm, and then we discuss soundness and completeness of RELAY.

### 3.1 RELAY

RELAY is a lockset-based static race detection tool that scales to millions lines of code. A `lockset` for a program point is the set of locks held at that point. A lockset-based analysis assumes that for every shared object there is at least one common lock that is held whenever that object is accessed. The tool reports a race if a pair of memory accesses in different threads could access the same shared object, the intersection of their locksets is empty, and at least one of the accesses is a write.

We briefly summarize RELAY's analysis, but details can be found in the original paper [50]. RELAY starts by analyzing every leaf function in the static call graph ignoring the calling context. For each leaf function, it computes a summary. A function's summary soundly approximates the effect of the function on the set of locks held before the function execution. Also, it includes a summary of the set of shared objects accessed in the function and the lockset held during each of its accesses. For example, a summary of a function `bar(void *b)` may say that a write to the field `b->bob` can happen while holding a lock `b->lock`, and that the function releases the lock `b->lock` before returning. RELAY composes function summaries in a bottom-up manner over the call graph by plugging in the summaries of the callee functions to compute the summaries of the callers.

Thus, RELAY performs a bottom-up calling-context-sensitive analysis on the call graph to compute the access summaries for all functions that are thread entry points. This is done using a combination of flow-insensitive points-to [3, 45] and symbolic analysis.

### 3.2 Soundness

Chimera only instruments data-races found by the static data-race detector. Therefore, its deterministic replay guarantees are based on the soundness of the static data-race detector it uses.

RELAY has three potential sources of unsoundness, but they are modularized and each one can be addressed separately using known techniques. First, RELAY ignores memory operations that occur inside blocks of assembly code when calculating lockset summaries. However, this issue could be addressed with additional engineering that integrates memory access analysis for assembly instructions [5] with RELAY, or via manual annotations.

Second, the points-to analysis [3, 45] used by RELAY does not handle pointer arithmetic. RELAY's pointer analysis assumes that after any arithmetic operation on a pointer, the pointer still points to the same object. When this assumption does not hold true, the pointer analysis is not guaranteed to be sound. As a result, we can guarantee replay for an execution only until the first buffer overflow. However, this does not fundamentally affect Chimera's analysis. Enhancing pointer analysis to handle pointer arithmetic [52], or ensuring language safety would address this problem.

Finally, RELAY post-processes data-race warnings using unsound filters, but we do not use them.

### 3.3 False Positives

To provide soundness, RELAY makes conservative assumptions, resulting in its reporting a high number of false data-races. Instrumenting weak-locks for every false data-race results in prohibitively high overhead.

There are two main sources of false positives. First, RELAY accounts for lock synchronizations, but ignores happens-before relationships due to non-mutex synchronization operations such as fork/join, barriers, and conditional variables. As a result, RELAY may report a data-race between memory operations that can never execute concurrently. The second main source of false positives is due to the conservative pointer analysis it uses [3, 45]. Conservative pointer analysis would cause RELAY to underestimate the lockset held by a code region and overestimate the variables that could be accessed by a memory instruction.

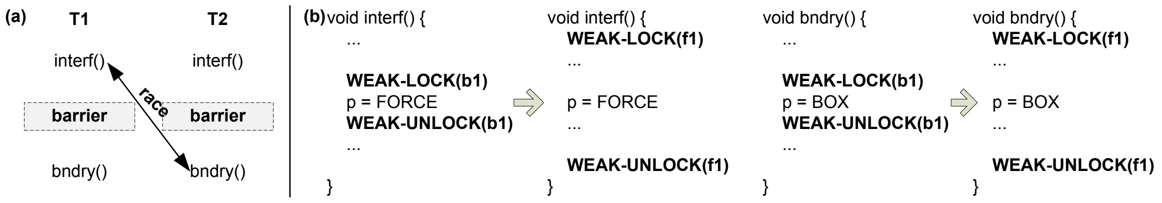
Our experimental results in Section 7 show that RELAY reported data-race warnings on about 14% of memory operations in a dynamic execution. Instrumenting them with weak-locks to record the order of those potential data-races incurs an approximately 53x slowdown. We next discuss two important optimizations based on profiling (Section 4) and symbolic bounds analysis (Section 5) that significantly reduce this cost.

## 4. Profiling Non-Concurrent Functions

A static data-race detector may report races between code regions that are never executed concurrently. One reason for this is the inadequacy of the static analysis in accounting for non-mutex synchronization operations. To address this issue, Chimera uses a profile-guided analysis to determine code regions that are likely to never execute concurrently and use that information to increase the granularity of weak-locks without compromising an application's parallelism.

### 4.1 Overview

One important limitation of lockset based static data-race detectors, including RELAY, is that they account only for locks, but ignore happens-before relations due to non-mutex synchronization operations. Many false data-races may be reported due to this limitation. Figure 2(a) illustrates a false data-race reported for `water`. The data-race is false because the two supposedly racy functions are never executed concurrently due to a barrier synchronization. We also find that a number of false data-races are reported between initialization code and the rest of the code regions, as RELAY does not account for fork-join synchronization. Another source of false data-races, unrelated to non-mutex synchronizations, is the lack of static knowledge of control dependencies. For example, we found



```
void interf() {        void interf() {        void bndry() {         void bndry() {
  ...                    WEAK-LOCK(f1)          ...                    WEAK-LOCK(f1)
                         ...                                           ...
  WEAK-LOCK(b1)                                 WEAK-LOCK(b1)
  p = FORCE              p = FORCE              p = BOX                p = BOX
  WEAK-UNLOCK(b1)        ...                    WEAK-UNLOCK(b1)        ...
  ...                                           ...
                         WEAK-UNLOCK(f1)                               WEAK-UNLOCK(f1)
}                      }                      }                      }
```

**Figure 2.** (a) A false data-race reported for `water` application from the SPLASH benchmark [53]. Functions `bndry()` and `interf()` are never executed concurrently due to the barrier synchronization, which is not accounted for in RELAY. (b) The granularity of weak-locks is increased to function level in the two potentially racy functions because Chimera's profiler finds them to be non-concurrent.

instances where a set of code regions are executed in only one thread, but RELAY reported false races among them. In all these cases, the two code regions containing the race-pair reported by RELAY are never executed concurrently.

We observe that such cases can be determined by profiling with a set of representative inputs. If a pair of potentially racy code regions are never executed concurrently in any of the profile runs, then there is sufficient confidence that they are likely to be *non-concurrent* in another execution. Profiling cannot guarantee that they will be non-concurrent in all executions. Nevertheless, we can take advantage of profiled information to increase the granularity of weak-locks to larger code regions and reduce the dynamic number of weak-lock operations.

If a pair of code regions containing a potential race-pair is likely to be non-concurrent, then Chimera increases the granularity of the weak-lock to protect the entire code region instead of just the basic blocks containing the race-pair. Figure 2(b) shows how this optimization affects the weak-lock instrumented to handle the false data-race that we discussed for `water` (Figure 2(a)). In this study, we consider functions as code regions while performing non-concurrent region profiling, but our method could be applied for other region granularities as well. We refer to a weak-lock that protects a function as a *function-lock*.

By increasing the granularity of the weak-lock to the function-level, Chimera reduces the dynamic number of operations on that lock. Increasing the granularity in terms of the code region size for a weak-lock also creates the opportunity to use a single weak-lock to guard multiple potential data-races. The next section discusses an optimization that exploits this opportunity.

### 4.2 Clique analysis

We propose a clique analysis to determine which racy function-pairs can share the same function-lock. Sharing a function-lock reduces the cost of instrumentation.

Figure 3(a) shows a graph with a node for every function that contains at least one potential data-race. A dotted edge connects a pair of functions that could potentially race. A solid edge connects a pair of functions that are found to be non-concurrent in all of the profile runs. For example, `alice` is potentially racy and non-concurrent with `bob` and `carol`. Functions `bob` and `carol` are non-concurrent, but are proven to be race-free with each other. Functions `bob` and `dave` are racy and have also been found to be concurrent in some profile run.

One simple algorithm would be to assign a unique weak-lock for every racy-function-pair. If the race-function pair is also non-concurrent, then we can use a function-level lock as shown in Figure 3(a). Note that `bob` and `dave` could run concurrently, and so we do not use function-level weak-locks to guard potential races between them, as that could serialize those the two concurrent functions and compromise on parallelism. Instead, a weak-lock is instrumented at the basic-block granularity.

The above algorithm requires that `alice` acquires and releases two function-level weak-locks (`f1 and f2`) every time it is executed. However, `alice`, `bob`, and `carol` are potentially non-concurrent with each other. Therefore, the two potential races could be guarded using a single function-lock `f0` as shown in Figure 3(b). This optimization would reduce the number of weak-lock operations.

To identify a group of functions which are mutually non-concurrent, we construct maximal cliques using a greedy algorithm in a graph of potentially non-concurrent functions (determined through profiling). A clique of an undirected graph is a subset of nodes where every node is connected to every other node. A maximal clique is a clique that cannot be extended by including one more adjacent node. Figure 3(c) shows a graph of potentially non-concurrent functions with two cliques, {`alice`,`bob`,`carol`} and {`carol`,`dave`}.

Once cliques are identified in a graph of non-concurrent functions, Chimera assigns function-locks as follows. For each race-pair, it checks if its racy functions are non-concurrent. If they are, then it finds the clique that the racy-function-pair is part of in the graph of non-concurrent functions. Chimera assigns the function-lock corresponding to that clique to both racy functions. For example in Figure 3(b), racy-function pairs {`alice`,`carol`} and {`alice`,`bob`} are both assigned a single function-lock `f0`. Notice that this weak-lock assignment is efficient for `alice` as it now has to acquire only one weak-lock as opposed to two. However, `bob` and `carol` are unnecessarily serialized (as they do not race with each other), which is still acceptable as they are also found to be non-concurrent during profiling.

It is possible that a racy-function-pair is part of two cliques. In that case, we use a greedy algorithm that chooses the weak-lock corresponding to the clique that contains the most number of racy-function-pairs.

## 5. Symbolic Bounds Analysis for Loops

Chimera's second optimization targets race-pairs that remain after applying the profile-based analysis described in the previous section. This optimization is based on symbolic address bounds analysis. It addresses the imprecision of the conservative but sound pointer analysis used in a static data-race detector.

### 5.1 Overview

Static data-race detectors [26, 50] use pointer analysis to determine the set of objects a memory instruction can access and also to determine the lockset at a program point. RELAY uses a combination of Steensgaard [45] and Andersen [3] flow-insensitive and context-insensitive pointer analysis, which are used in many static tools because they scale well to large programs. However, because these analyses are very conservative, RELAY overestimates the range of addresses that a memory instruction can access and underestimates the set of locks held at a program point, both of which cause it to report a number of false data races.

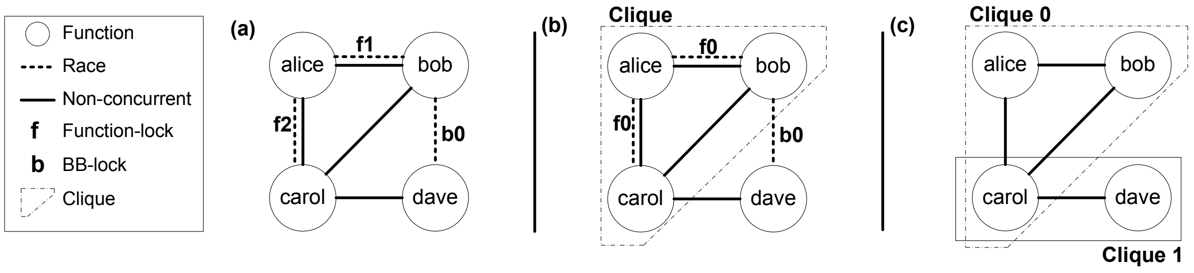


**Figure 3.** (a) One weak-lock is instrumented for each race-pair. If a racy function-pair is non-concurrent, a function-level weak-lock (`f1`, `f2`) is used. Otherwise, a basic-block level weak-lock is used (`b0`). (b) Two potential data-races in a clique in a graph of non-concurrent function share one function-lock (`f0`). (c) Cliques in a graph representing non-concurrent functions.

```
1:  for( i = 0 ; i < max_digits ; i ++ ) {
2:     WEAK-LOCK (&rank[0] to &rank[radix-1]);
3:     for ( j = 0 ; j < radix ; j++ ){
4:        rank[ j ] = 0;
5:     }
6:     WEAK-UNLOCK (&rank[0] to &rank[radix-1]);
7:
8:     WEAK-LOCK (-INF to +INF);
9:     for ( j = start ; j < stop ; j ++ ) {
10:       my_key = key_from[ j ] & bb;
11:       rank[ my_key ]++;
12:    }
13:    WEAK-UNLOCK (-INF to +INF);
14: }
```

**Figure 4.** Instrumenting weak-locks for a loop in the function `slave_sort` in `radix` using symbolic bounds.

For example, we find a number of false data-races between two functions executed concurrently in different threads. This often happens when a programmer partitions work between threads, but the static analysis is unable to determine that the function will access different parts of a data structure. Figure 4 shows an example. RELAY reports a false data-race on the `rank` array in line 4 and 11, and also on `key_from` array in line 10. However, `radix` divides a large array into multiple portions and assigns different portions to concurrent threads to process them in parallel. Therefore, the base address of `rank` and `key_from` are different for each worker thread, and hence the threads do not access the same entry in those arrays concurrently.

It is hard to statically determine the absolute values of address bounds of an object accessed by memory operations in a code region. However, it has been shown that the lower and upper bounds in the form of a symbolic expression can often be derived statically [41, 42] with much better accuracy. Chimera uses this information to increase the granularity of weak-locks that it must instrument for race-pairs in concurrent code regions. The weak-lock is constructed in such a way that it protects a code region for a range of addresses specified by a symbolic expression. Thus, two potentially racy code regions can execute concurrently (provided our symbolic bounds are accurate enough). At the same time, Chimera can protect the regions with weak-locks instrumented at larger granularities to reduce the number of operations.

Figure 4 shows an example. RELAY reports that `line 4` could race with itself. Instrumenting a weak-lock inside the loop would be very expensive. Instead, Chimera instruments a weak-lock that provides mutual exclusion for the entire loop (lines 3-5) only for the address range from `&rank[0]` to `&rank[radix-1]`. This range is computed by a sound static symbolic address bounds analysis, which we discuss in the next section.

## 5.2 Symbolic Bounds Analysis

We implemented our symbolic bounds analysis based on the algorithm proposed by Rugina and Rinard [41, 42]. The goal of this analysis is to determine the symbolic expressions that specify the upper and lower bounds for a pointer or array index variable at a program point that is found to be potentially racy by the static data race detector. For the example in Figure 4, the analysis determines that the symbolic lower bound of $j$ of the first inner loop (line 4-7) is 0 and the upper bound is the initial value of radix $radix_0 - 1$. It also finds that `line 4` can access a memory region from `&rank[0]` to `&rank[radix-1]`. Details about the algorithm are discussed by Rugina and Rinard [42].

The effectiveness of our optimization depends on the accuracy of the lower and upper bounds. The analysis we use is sound, but imprecise. If the bounds are too conservative, we may serialize concurrent code regions unnecessarily. There are two main sources of imprecision. The first case is when the address of the racy object is based on the value of a variable that cannot be determined outside the code region. For example, precise symbolic bounds for the `rank` array accesses in the second inner loop (line 9-12) cannot be determined. The value of the index variable $my\_key$ cannot be computed outside the loop as it depends on the value read from another array $key\_from$ inside the loop (line 11). However, we can derive the symbolic bounds for the array $key\_from$ accurately. The second source of inaccuracy is when the racy object's bounds depends on an arithmetic operation (e.g., the modulo operation or logical AND/OR) not supported in the analysis.

## 5.3 Choosing the Granularity for Code Region

Rugina and Rinard's analysis [41, 42] describes a generic algorithm for larger code regions including inter-procedural analysis, but, as a first step, we applied their technique only for loops with no function calls in the loop body. As a result, our current implementation may not exploit all opportunities for optimization.

If the symbolic bounds are too imprecise, care must be taken to ensure that we do not over-serialize loops. If the derived symbolic expression for an address range is from negative infinity to positive infinity, we consider it to be *too imprecise* to be useful. Otherwise, we consider it to be *precise enough*. In that case, we balance the number of weak-lock operations with the loop serialization cost.

If the symbolic bounds of a racy loop is precise enough, we assign a weak-lock at the loop granularity (the first inner loop in Figure 4). If the bounds are too imprecise, we estimate via profiling the average number of instructions executed by a loop iteration. If the estimate is less than a `loop-body-threshold`, we still instrument at the loop granularity because the cost of operations on the weak-lock does not warrant exposing parallelism in the loop. Otherwise, we instrument a basic-block lock inside the loop body.

If a loop is nested, we select the outermost loop with precise enough bounds.

## 6. Implementation

This section presents the implementation details of the Chimera record and replay system.

### 6.1 Analysis, Instrumentation, and Runtime System

Our analysis and instrumentation framework is implemented in OCaml, using CIL [35] as a front end. To profile concurrent function pairs (Section 4), we instrumented the entry and exit of each function using CIL's source-to-source translation. To statically derive symbolic bounds of racy loops (Section 5), we also performed data flow analysis on a racy loop and produced linear programming constraints using CIL. Then, we used `lpsolve` [1], a mixed integer linear programming solver, to find a solution for static bounds that a racy loop may access. Finally, based on the results of the above static analysis, we used CIL to instrument weak-locks at the function, loop, basic block, or instruction granularity.

We modified the Linux kernel to record and replay nondeterministic input from system calls and signals. We also modified GNU pthread library version 2.5.1 to log the happens-before order of the original synchronization operations and the weak-locks added by Chimera.

### 6.2 Static Analysis and Source code

We used RELAY [50] to perform pointer analysis and to collect a set of potential data-races. We applied Andersen's inclusion-based pointer analysis [3] to resolve function pointers, and Steensgaard's unification-based approach [45] to perform alias analysis between lvalues. While performing pointer analysis, RELAY first translates function local arrays and address-taken variables to heap variables (making them global) in order to derive pointer constraints in a unified manner. RELAY performs static analysis on this modified source code. This can lead to unnecessary false data-races on local variables. To resolve this, we filtered out race warnings on a 'heapified' local variable that did not escape its function.

To perform sound static analysis, we made sure that all library source code (except for `apache` and `pbzip2`) are included in our static analysis. For the standard C library, we used uClibc [48] which is smaller and easier to analyze than the GNU glibc library, as it is developed for embedded Linux systems. The uClibc library involves all the necessary functions such as `libc` and `libm`.

For `apache`, we did not include libraries such as `gdbm`, `sqlite3`, etc., because they do not contain code that gets executed for the input we use in our study. It is possible that the source code of a third party library may not be available for static analysis. When any part of the source code of a library used by a program is not analyzed, the soundness of static analysis may be compromised. One solution is to ask library builders to provide annotation (lockset summaries) for their library functions so that it can be fed into RELAY to perform a sound data-race analysis. Developing such annotations would be an one-time cost for library builders, and it would not place any burden on software developers that use those libraries. Another possible solution is to assume that a library function will only access the set of objects pointed to by the parameters passed as function arguments without acquring any new locks. However, this approach is not guaranteed to be sound, because a library could retain pointers passed to previous calls to the same library. Also, instructions in a library's function can have a data-race on some shared-variable that is internal to the library. We employed the latter approach for `pbzip2` (we excluded the `libbz2` library used by `pbzip2`)

We also converted the C++ `pbzip2` program into ANSI-C code by replacing the vector STL container with a linked-list-based C library, because our instrumentation framework, CIL [35], can only handle C programs, but not C++ constructs.

## 7. Results

This section evaluates Chimera's recording and replaying overhead and demonstrates the effectiveness of the profiling and symbolic bounds optimizations.

### 7.1 Methodology

We evaluated our system using three sets of benchmarks which are listed in Table 1. The first set consists of three desktop applications: `aget`, `pfscan`, and `pbzip2`. The second set has two web sever programs: `knot` and `apache`, which are evaluated using the ApacheBench (ab) client. The final set contains four scientific programs from SPLASH-2 [53]: `ocean`, `water`, `fft`, and `radix`. To collect a set of concurrent function pairs for clique analysis (Section 4.2), we profiled each program 20 times with various inputs. The inputs used for profiling are significantly different from the input used for our performance evaluation.

Chimera is scalable to large programs. It is built on RELAY [50], which has been shown to scale to very large programs (e.g., Linux with 4.5 million lines of code). Chimera also uses static analysis to derive symbolic bounds, but it is a scalable intraprocedural analysis. Our benchmark set includes some fairly large programs. Table 1 provides the number of lines (LOC) of our benchmarks in their CIL representation. It does not account for the size of library code: libc(41.7K) and libm(3.6K).

Presence of assembly code and buffer overflow may compromise the soundness of static data-race analysis (Section 3.2). However, the programs we evaluated do not contain assembly code, except for a few library functions. Also, we are not aware of any buffer overflow bugs in our benchmarks. Also, we did not observe any weak-lock timeouts (Section 2.3) in any of our experiments.

We ran our experiments on a 2.66 GHz 8-core Xeon processor with 4 GB of RAM running CentOS Linux version 5.3. We modified Linux 2.6.26 kernel and GNU `pthread` library version 2.5.1 to support Chimera's record and replay features. All results are the mean of five trials with 4 worker threads (excluding main or control threads). Section 7.2 presents scalability results for which we used 2, 4, and 8 threads.

### 7.2 Record and replay performance

Table 2 shows Chimera's record and replay performance when all the optimizations (function, loop, and basic-block level weak-lock optimizations) are enabled. The first set of columns quantifies the number of logs generated for recording program input (read through systems calls) and the happens-before order of synchronization operations. These logs are sufficient to guarantee replay for data-race-free (DRF) programs. The second set of columns quantifies the number of logs due to various types of weak-locks. The next set of columns presents the performance overhead. The last set of columns quantifies the `gzip` compressed log sizes for recording the program input and the order of all synchronization operations (including weak-locks).

Chimera incurs negligible overhead for desktop and server applications. For scientific applications (with high frequency of accesses to shared variables) the overhead is relatively high. On average, our system incurs 40% performance overhead to record an execution with four worker threads. Replay overhead is similar to that of recording overhead for most applications, except for I/O intensive applications. Network intensive applications such as `aget`, `knot`, and `apache` replay much faster as we feed the recorded input directly to the replayed process without waiting for the network response. Chimera's performance overhead is an order of magnitude

| application | | LOC | profile environment | evaluation environment |
|---|---|---|---|---|
| desktop | aget | 1.2K | 2 workers, download a 29KB file from local network | 2,4,8 workers, download a 10MB file from http://ftp.gnu.org |
| | pfscan | 2.1K | 2 workers, scan 236 KB of small 22 files | 2,4,8 workers, scan 952 MB of 8 log files |
| | pbzip2 | 4.8K | 2 workers, compress a 219 KB file, output to stdout | 2,4,8 workers, compress 16 MB file, output to file |
| server | knot | 1.3K | 2 workers, 4 clients, 100 requests, 29KB file | 2,4,8 workers, 16 clients, 1000 requests, 390KB file |
| | apache | 99K | 2 workers, 4 clients, 100 requests, 29KB file | 2,4,8 workers, 16 clients, 1000 requests, 390KB file |
| scientific | ocean | 5.3K | 2 workers, 130*130 grid, 1e-01 error tolerance | 2,4,8 workers, 1026*1026 grid, 1e-07 error tolerance |
| | water | 2.5K | 2 workers, 64 molecules, 5 steps | 2,4,8 workers, 1000 molecules, 10 steps |
| | fft | 1.4K | 2 workers, $2^4$ matrix , no inverse FFT check | 2,4,8 workers, $2^{20}$ matrix, with inverse FFT check |
| | radix | 1.3K | 2 workers, $2^8$ keys , no sanity check | 2,4,8 workers, $2^{14}$ keys, with sanity check |

**Table 1.** Benchmarks and input used for profiling and evaluating Chimera. The number of lines in the source program (LOC) is measured for the CIL representation. It does not include the size of library code: libc(41.7K) and libm(3.6K).

| application | | DRF Logs | | logging order of potential data-races | | | | performance | | | | log size | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | system calls | synch. ops. | instr. log | basic blk. log | loop log | func. log | original time(ms) | record time(ms) | recording overhead | replay overhead | input log(KB) | order log (KB) |
| desktop | aget | 16604 | 8424 | 28876 | 5191 | 15939 | 32416 | 5058 | 5114 | 1.01 | 0.06 | 20072 | 361 |
| | pfscan | 109 | 879 | 8 | 0 | 39 | 347 | 848 | 881 | 1.04 | 1.02 | 2 | 3 |
| | pbzip2 | 592 | 2491 | 2621 | 81 | 1177 | 1540 | 1343 | 1371 | 1.02 | 1.03 | 1989 | 26 |
| server | knot | 8056 | 32 | 5136 | 0 | 251 | 2257 | 7137 | 7176 | 1.01 | 0.01 | 84 | 23 |
| | apache | 18301 | 36812 | 798891 | 266956 | 565863 | 1123337 | 18668 | 19376 | 1.04 | 0.02 | 178 | 6469 |
| scientific | ocean | 2750 | 9978 | 6237 | 8233 | 287642 | 37655 | 2328 | 5585 | 2.40 | 2.24 | 16 | 727 |
| | water | 10295 | 67202 | 21838 | 1409884 | 198993 | 1112798 | 1665 | 2820 | 1.69 | 1.75 | 101 | 12744 |
| | fft | 113 | 193 | 1843 | 38 | 49718 | 11595 | 586 | 1249 | 2.13 | 2.23 | 2 | 107 |
| | radix | 102 | 312 | 3 | 13 | 344 | 393 | 1599 | 1939 | 1.21 | 1.20 | 1 | 3 |

**Table 2.** Chimera record and replay performance. The results are the mean of five trials with 4 worker threads.

improvement over the state-of-the art software solutions that guarantee multiprocessor replay [49].

Log sizes of Chimera are within acceptable limits for various uses of replay. `aget` produces large logs because the contents of all the downloaded files are in the log. `water` also produces a large log size because of frequent user specified synchronizations and weak-locks.

## 7.3 Effectiveness of Optimizations

We analyzed the effect of different optimizations on recorder's overhead. Fine grained weak-locks (instruction and basic-block level weak-locks) enable higher concurrency, but they increase the number of program points instrumented resulting in higher performance and log size overhead. The opposite is true for coarser grained weak-locks such as function and loop level weak-locks. We use function-level weak-locks if two functions are likely to be non-concurrent (Section 4). We use loop-level weak-locks with runtime bounds checks if our static analysis can derive precise enough symbolic bounds (Section 5).

Figure 5 shows the performance overhead of Chimera's recorder with different sets of optimizations normalized to native execution time. As expected, instrumenting every potential data-race at the granularity of a source line (labeled as 'instr') incurs 53x slowdown. However, when we apply the profile-based optimization to increase the granularity of some weak-locks to function level ('inst+func') the overhead drops to 27x. If we use only symbolic analysis to coarsen the granularity of some weak-locks to loop level results in 33x overhead. However, when we employ all the optimizations together ('inst+bb+loop+func'), including basic block level weak-locks, the average overhead drops significantly to 1.39x.

Applications such as `pfscan` and `water` benefit significantly from function-level locks. In these applications, most data-races are in function-pairs ordered by some non-mutex synchronization operations that our static analysis could not account for. For applications such as `apache`, `ocean`, `fft`, and `radix`, loop-level locks reduce the recording overhead drastically. For example, in `apache`, RELAY reports a false data-race between memory operations within a hot loop in the `memset` library function that iterates approximately over 6 million times in our experiments. Function-level weak-lock is ineffective in this case, because two threads may execute the `memset` concurrently. However, our static analysis determines the bounds of addresses accessed within the hot loops of `memset` fairly accurately, which enabled us to use loop-level weak-locks effectively. We also observe noticeable benefits in coarsening weak-locks from instruction-level to basic-blocks (e.g., `water`).

Finally, for network applications like `aget`, `knot`, and `apache`, recording cost overlaps with I/O wait resulting in negligible overhead. Chimera could be used even in production systems for such applications.

Figure 6 shows the proportion of dynamic number of weak-lock operations with respect to the total number of dynamic memory operations. A naive dynamic data-race detector would have to instrument 100% of memory operations. This result shows the advantage of static data-race analysis and our optimizations in terms of reducing the number of instrumented points in the program. In general, results in Figure 6 for different optimizations are consistent with the recording overhead in Figure 5. This indicates that the savings obtained from coarser weak-locks was not overshadowed by any loss in parallelism.

On average, naively monitoring all data-races reported by the static data-race detector requires us to instrument about 14% of all dynamic memory operations. By increasing the weak-lock granularity to function, loop, and basic block levels, we can reduce the proportion of weak-lock operations with respect to memory accesses down to 0.02% on average. In general, this result shows that

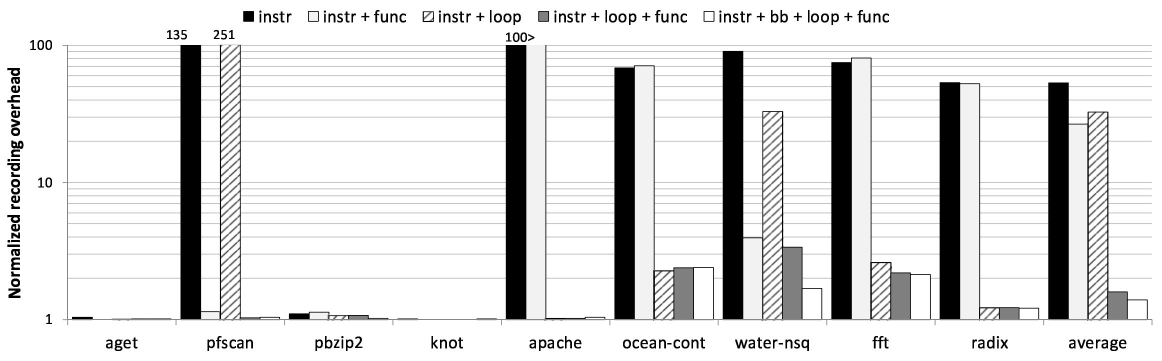


**Figure 5.** Normalized recording overhead for Chimera with different sets of optimizations

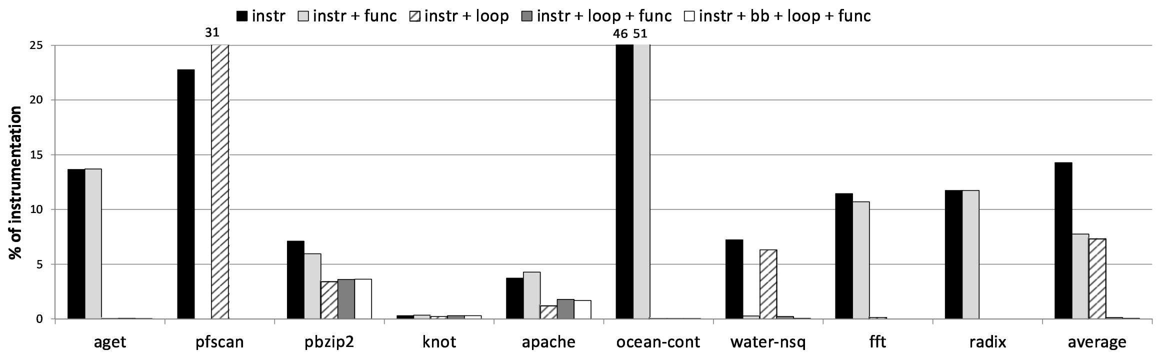


**Figure 6.** Proportion of instrumentation points for different logging schemes

increasing the granularity of weak-locks reduces the instrumentation cost.

However, in some fairly rare cases, increasing the locking granularity from instruction to loop-level may increase the frequency of weak-lock operations. The reason is due to control flow dependencies. In `pfscan`, there is a racy instruction inside a hot loop that is guarded by an `if` statement. If we use loop-level weak-lock, Chimera has to always perform weak-lock operations when the loop is executed. But if we use instruction-level weak-lock, the instrumented code will be executed only when the `if` condition gets satisfied.

In `apache` the number of weak-lock operations increase when we go from instruction-level to function-level granularity. The reason for this is behavior is best explained using a contrived example. Assume that RELAY finds a data-race between each of the functions `foo`, `bar`, and `qux`. Also, assume that all these functions are non-concurrent with each other, except for the function-pair `bar` and `qux`. For this example, Chimera will assign two different function level weak-locks (one for `foo-bar` and another for `foo-qux`). This allows `bar` and `qux` to run concurrently. As a result, `foo` is instrumented with two function-level locks, which may be more costlier than using one instruction-level lock if there is only one racy instruction inside `foo`.

We also studied the sensitivity of our profile-based non-concurrent function analysis to the number of profile runs. We did this study only for `pfscan` and `water-nsq`, because other applications shows little performance benefit from function-level logging (Figure 5). For these two applications, the number of concurrent function pairs observed quickly saturates after a small number of profile runs (five for `pfscan` and three for `water-nsq`).

## 7.4 Sources of Overhead and Scalability

Figure 7 provides a breakdown of the remaining sources of performance overhead in the Chimera recorder that incorporates all of our optimizations ('inst+bb+loop+func'). The results are normalized to the native execution time. We measure the performance of our system by instrumenting each type of weak-lock one by one. The performance overhead due to a weak-lock type is further broken down into the cost of logging the weak-lock operations and the cost due to weak-lock contention. To measure the time lost due to weak-lock contention, we subtracted the execution time of a program execution in which a weak-lock acquire operation always succeeds without waiting from the execution time of a program execution in which the weak-locks semantics are obeyed.

Contention for loop-level weak-locks dominate the overhead for scientific applications such as `ocean` and `fft`. The reason is that our static symbolic bounds analysis is not very precise for some performance critical loops in these programs because they tend to execute irregular array accesses and unmodeled arithmetic operations (Section 5.2). As a result, the bounds checks performed as part of loop-lock acquire operation over-serializes the execution. We also fail to use loop-level lock and resort to instruction-level logging (e.g., `water`), if the loop body contains a function call, because our symbolic analysis in intra-procedural.

Contention between loop-level weak-locks is the for increase in performance overhead as the number of threads increases for some applications (Figure 8). We believe that source-level inlining for small functions or inter-procedural symbolic bounds analysis could help reduce this overhead. Nevertheless, as we discussed earlier (Section 7.3), our current analysis already provides significant benefits with loop-level locks for many applications (e.g., `ocean`).

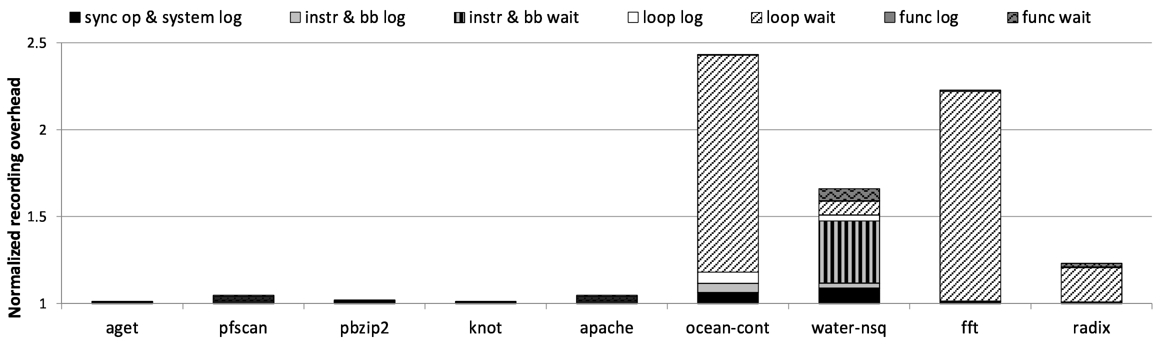


**Figure 7.** Sources of recording overhead

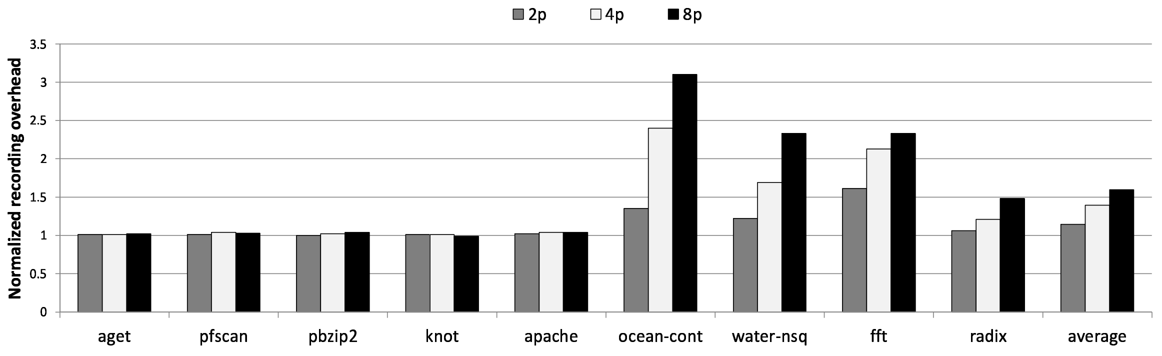


**Figure 8.** Scalability results on 2,4,and 8 processor executions

## 8. Related Work

Chimera is related to three distinct areas of prior work: deterministic record-and-replay, deterministic execution, and hybrid data-race detection.

**Deterministic Replay.** Support for checkpointing and logging non-deterministic input (interrupts, DMA, I/O, etc.) is sufficient to guarantee deterministic replay for single-threaded programs [44]. Deterministic replay for multithreaded programs running on a uniprocessor also can be provided at low cost by recording thread schedules [16]. However, multiprocessor replay remains an open challenge due to difficulties in detecting and logging shared memory dependencies. The earliest work in this area either incurred prohibitive performance overhead due to the cost of monitoring memory operations [8, 17, 34], supported deterministic record and replay only for data-race-free programs [40], or relied on custom hardware support [33, 54].

State-of-the-art software solutions mainly take two approaches: search or redundancy. PRES [38] and ODR [2] are offline search based replay systems. Instead of detecting and recording shared memory dependencies at runtime, they perform offline search to reconstruct thread interleavings. This class of systems show notable performance improvement during recording, but the offline search is not guaranteed to succeed in a bounded amount of time (failing to find a solution in some experiments). In some cases, the first deterministic replay may take prohibitively long time when the search does not scale. However, subsequent replays have low overhead, because a solution has been previously found. Alternatively, Respec [29] and DoublePlay [49] use redundant execution to detect data-races during recording that could compromise deterministic replay. While the record and replay latency overhead is low when spare cores are available to run an additional replica, these systems impose a minimum of a 2x CPU throughput overhead to run an additional replica.

LEAP [25] uses static escape analysis to provide efficient multiprocessor replay. LEAP improves the efficiency of a recorder by instrumenting accesses to only shared variables that are determined using a static escape analysis. LEAP also ignores accesses to variables that are immutable after initialization to improve efficiency. Monitoring and logging accesses to all mutable shared variables determined using a conservative static analysis can be quite expensive at runtime. LEAP can slowdown a program by more than 2x in the average case and 6x in the worst case [25].

In contrast to these prior systems, Chimera uses a sound static data race analysis and a series of optimizations to build the most efficient software replay solution for commodity multiprocessors to date.

**Deterministic Execution.** Deterministic execution systems help programmers reproduce a multiprocessor execution by ensuring that the thread interleaving observed is always the same for a given program and an input [7, 9, 14, 37]. This approach obviates the need for recording the order of shared-memory accesses, but must still record any non-deterministic program input to provide deterministic replay. While deterministic execution can be supported fairly efficiently for programs without data-races [9, 37], software only solutions for racy programs incur many orders of slowdown [6, 30]. Efficiency can be improved by either using custom hardware [14, 15, 23], or by restricting the class of programs supported to fork-join parallelism [7] or shared-nothing address spaces [4]. Chimera transforms a program into an equivalent data-race-free program under the new set of synchronization operations. Future work can leverage this property to design an efficient software only solution for deterministic execution.

**Data-Race Detection.** There is a large body of work that uses locksets to perform static data-race detection for C/C++ programs [19, 26, 39, 46]. Type systems have been used to improve static data-race analysis [10, 21, 22]. We used lockset based RELAY [50] to build Chimera, but future advancements in this area could help us further reduce the overhead of our replay system.

Perhaps the most closely related study is the work on hybrid data-race detectors that used static analysis to eliminate runtime checks for memory operations that are proven to be data-race-free [12, 18]. Unlike Chimera, they check all suspected racy accesses at the instruction granularity, which we show could lead to a high runtime overhead. To reduce this overhead, Choi et al. [12] discuss unsound optimizations that may not find more than one data-race per memory location. Such weaker guarantees may be acceptable for detecting concurrency bugs, but are not sufficient to guarantee deterministic replay.

## 9. Conclusion

Non-determinism has been one of the thorny issues in shared-memory multithreaded programming. An efficient deterministic replay system can help solve this problem by empowering programmers with the ability to reproduce and understand a program's execution. This is critical in many stages of the software development process, including debugging, testing, reproducing problems from the field, and forensic analysis.

Chimera is the first software system for multiprocessors that leverages a static data-race detector tool to provide a low overhead replay solution. However, an efficient solution would not have been possible without the two critical optimizations that we employed to drastically reduce the overhead of recording all the data-races reported by a conservative, but sound static analysis tool.

Chimera's transformations ensure that the resultant code is data-race-free when instrumented with the new set of synchronization operations. We believe that this technique could also prove quite useful for enabling stronger semantics for concurrent languages such as sequential consistency and for enabling deterministic execution.

## Acknowledgments

This work was funded in part by NSF with grants CCF-0916770, CNS-0905149, a Microsoft gift and an equipment grant from Intel.

## References

[1] lpsolve, mixed integer linear programming solver. http://lpsolve.sourceforge.net/5.5.

[2] G. Altekar and I. Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 193–206, October 2009.

[3] L. O. Andersen. Program analysis and specialization for the c programming language. In *PhD thesis, DIKU, University of Copenhagen*, 1994.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, Vancouver, BC, 2010.

[5] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *In CC*, pages 5–23. Springer-Verlag, 2004.

[6] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–64, Pittsburgh, PA, 2010.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 81–96, Orlando, FL, October 2009.

[8] S. Bhansali, W. Chen, S. de Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, pages 154–163, June 2006.

[9] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 97–116, Orlando, FL, October 2009.

[10] C. Boyapati and M. Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 56–69, New York, NY, USA, 2001.

[11] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[12] 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 *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[13] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Technical Conference*, pages 1–14, June 2008.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, March 2009.

[15] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 67–78, 2011.

[16] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, MA, December 2002.

[17] G. W. Dunlap, D. G. Lucchetti, M. Fetterman, and P. M. Chen. Execution replay on multiprocessor virtual machines. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 121–130, March 2008.

[18] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.

[19] D. Engler and K. Ashcraft. RacerX: Efficient static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, NY, 2003.

[20] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 121–133, Dublin, Ireland, June 2009.

[21] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 219–232, Vancouver, British Columbia, Canada, 2000.

[22] D. Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 13–25, New York, NY, USA, 2003.

[23] D. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or not? free will to choose. In *17th International Conference on High-Performance Computer Architecture*, HPCA '11.

[24] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 2008 Inter-*

*national Symposium on Computer Architecture*, pages 265–276, June 2008.

[25] R. Huang, D. Y. Den, and G. E. Suh. Orthrus: Efficient software integrity protection on multi-cores. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 371–383, Pittsburgh, PA, March 2010.

[26] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 13–22, New York, NY, USA, 2009.

[27] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 1–15, April 2005.

[28] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4): 471–482, 1987.

[29] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–89, Pittsburgh, PA, March 2010.

[30] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, 2011.

[31] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 277–288, Beijing, China, 2008.

[32] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 2008 International Symposium on Computer Architecture*, pages 289–300, June 2008.

[33] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using Strata. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, 2006.

[34] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, pages 216–227, June 2006.

[35] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, 2002.

[36] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, Seattle, WA, March 2008.

[37] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 2009 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–108, March 2009.

[38] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd SOSP*, pages 177–191, October 2009.

[39] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33:3:1–3:55, January 2011.

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

[41] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 182–195, New York, NY, USA, 2000.

[42] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst.*, 27:185–235, March 2005.

[43] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 258–266, 1996.

[44] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, pages 29–44, Boston, MA, June 2004.

[45] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996.

[46] N. Sterling. Warlock: A static data race analysis tool. pages 97–106, 1993.

[47] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 131–144, October 2007.

[48] uclib.org. uClibc, a C library for embedded Linux. `http://uClibc.org`.

[49] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, March 2011.

[50] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, Dubrovnik, Croatia, 2007.

[51] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the 2010 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 155–166, March 2010.

[52] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995.

[53] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[54] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 2003 International Symposium on Computer Architecture*, pages 122–135, June 2003.

[55] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 2007 Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2007.

[56] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*, pages 321–334, April 2010.