REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
31-03-2012	Final Report	Mar 2009 – Dec 2011

4. TITLE AND SUBTITLE

(YIP-09) SECURE HETEROGENOUS MULTICORE PLATFORM THROUGH DIVERSITY AND REDUNDANCY

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA9550-09-1-0131

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Suh, Gookwon, E.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Cornell University
Ithaca, NY 14853

8. PERFORMING ORGANIZATION REPORT NUMBER

OSP 57093

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research
875 North Randolph Street
Suite 325, Rm 3112
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2012-0988

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution A: Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project aimed to significantly enhance the security of future multi-core platforms against software exploits by exploiting abundant parallel computing resources. In this context, the project developed hardware-assisted run-time monitoring techniques to detect attacks exploiting memory safety errors or emerging parallel program errors. The project also found that security and reliability techniques could be combined, in particular in the context of off-chip memory protection, to provide both properties with the cost of one. In addition to developing efficient and effective run-time protection techniques against software exploits, this project also resulted in multiple contributions in the general area of building more secure hardware foundations, including a secure hardware design process, hybrid memory with an application to fast recovery, hardware device signatures, and true random number generation.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Robert Herklotz
U	U	U	U	25	19b. TELEPHONE NUMBER *(include area code)* 703-696-6565

Final Project Report

Title: Secure Heterogeneous Multicore Platform Through Diversity and Redundancy
PI: G. Edward Suh, Cornell University
Grant No.: FA9550-09-1-0131
Program Manager: Robert L. Herklotz

The goal of this project was to significantly enhance the software security of future computing platforms by exploiting abundant parallel computing resources in many-core platforms. The initial investigation focused on protecting software through efficient diversification and replication. Recent studies indicate that introduction of automatically generated diversity and redundancy can provide transparent and comprehensive protection against a large class of software exploits. For example, an attack that exploits memory safety bugs can be detected by running two replicas of a program with different memory layouts. However, execution of diverse replicas on todays computing platforms incurs significant overheads in memory space, power consumption and bandwidth consumption, which prevent the diverse replication from being widely deployed. This project developed a heterogeneous multi-core architecture that can enable simultaneous execution of multiple replicas with minimal overheads be exploiting unnecessary redundancies, and demonstrated a system that can detect both temporal and spatial memory errors with less than 20% overheads compared to over 2x overheads today.

The project expanded this general approach of hardware-assisted run-time detection to concurrency attacks on parallel programs. A recent study showed that concurrency bugs, which allow an unintended interleaving among threads, could be maliciously exploited to compromise a system that runs multi-threaded applications. In this project, the PI's team developed a new concurrency bug detection scheme that can detect a broad range of bugs, including both data races and non-race bugs. In particular, the study found that common non-race bugs in practice can be captured by checking critical sections that are not controlled through explicit communications. Experiments show that the proposed detection technique provides a significantly better coverage without introducing more false positives, compared to traditional race detectors. The team also developed hardware support for the new detection scheme and demonstrated that the check can be performed with only a few percent overheads.

This investigation also revealed that there exists a significant synergy between run-time techniques for security and reliability, which can be leveraged to reduce overheads when both are required. As an example, today, a system needs two independent mechanisms in order to protect the memory integrity from both physical attacks and random errors. Integrity verification schemes detect malicious tampering of memory while error correcting codes (ECC) detect and correct random errors. This project developed a unified off-chip memory integrity protection scheme that provides both detection of malicious attacks for security and correction of random errors for reliability at the same time by extending the integrity verification techniques. When both security and reliability are required, this technique effectively removes the memory and bandwidth overheads (12.5%) of typical ECC schemes.

In addition to developing efficient and effective run-time protection techniques against software exploits, this project also resulted in multiple contributions in the general area of building more secure hardware foundations. For example, the team worked with researchers at Intel to develop a new security assessment process for early hardware designs, introduced a new SRAM-eDRAM

hybrid memory structures that can quickly checkpoint data, and found that Flash memory can be leveraged as hardware-based security primitives.

Major Accomplishments:

- *Efficient Software Replication for Memory Safety*: This project developed a heterogeneous multi-core architecture, named Orthrus, which can protect a system against both temporal and spatial memory exploits with efficient exeuction of two replicas with different memory layouts. This work is detailed in the ASPLOS 2010 publication.

- *Run-Time Detection for Parallel Program Vulnerabilities*: This project developed a new parallel program bug detection technique along with efficient hardware support to enable the scheme in production systems. This work is being prepared for publication. The manuscript with technical details is attached at the end of this report.

- *Synergy between Security and Reliability*: This project demonstrated that security and reliability techniques can be combined together to reduce the overheads without sacrificing capabilities to deal with errors and attacks. This work is published in ISCA 2010.

- *Security Assessment Scheme for Architecture Features*: One critical aspect of a secure hardware design is the ability to measure a design's security. The PI's team worked with Intel researchers to study a systematic way of measuring and categorizing a hardware feature's security concern at an early design stage. This work is detailed in the publication in TRUST 2011.

- *SRAM-eDRAM Hybrid Memory*: This project partially supported a development of hybrid memory structure that combines SRAM and eDRAM within each memory cell. This technology can enable instant checkpointing and roll-back of microprocessor state with a potential application to rapid recovery from attacks. The memroy design has initially been applied to improve the energy consumption and area of GPGPUs (ISCA 2011 paper).

- *Signature and True Random Number Generation from Flash Memory*: The PI's team found that unmodified commercial Flash memory can provide two important hardware security functions: true random number generation and digital fingerprinting. These primitives provide strong hardware foundataions for secure systems. The details can be found in the publication in IEEE Security Privacy 2012.

Publications:

- Ruirui Huang, Daniel Y. Deng, and G. Edward Suh, Orthrus: Efficient Software Integrity Protection on Multi-Cores, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, March 2010.

- Ruirui Huang, and G. Edward Suh, IVEC: Off-Chip Memory Integrity Protection for Both Security and Reliability, *Proceedings of the 37th International Symposium on Computer Architecture (ISCA 2010)*, June 2010.

- Wing-kei Yu, Ruirui Huang, Sarah Xu, Sung-En Wang, Edwin Kan, and G. Edward Suh, SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading, *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, June 2011.

- Ruirui Huang, David Grawrock, David C. Doughty, G. Edward Suh, Systematic Security Assessment at an Early Processor Design Stage, *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST)*, June 2011.

- Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G. Edward Suh, and Edwin Kan, Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints, *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.

Supported Personnel:

- Gookwon Edward Suh (PI)
- Danny Y. Deng
- Ruirui Huang
- Daniel Lo
- Yao Wang
- Yinglei Wang

Run-Time Concurrency Bug Detection Through Uncontrolled Memory Accesses

1 Introduction

As computing hardware moves to multi-core and many-core systems, future software needs to be parallelized in order to benefit from the increasing computing resources. However, writing a correct parallel program is notoriously difficult, partly because of non-determinism in concurrent program executions. Because thread executions can be interleaved in many ways, a parallel program may produce a non-deterministic outcome even for identical program inputs if threads are not properly synchronized. Such a non-deterministic behavior, if not intentional, is often referred to as a concurrency bug.

In this paper, we propose to extend the intuition behind the traditional data races to develop a general scheme that can detect a broad range of concurrency bugs, both data races and non-race bugs, without relying on application-specific knowledge. A data race refers to conflicting (same location, at least one write) memory accesses from multiple threads that are not synchronized at all. In essence, this *lack of control by a programmer* is commonly considered as a sign of a bug. However, simply having a synchronization operation does not necessarily mean it is correct, and recent studies show that many concurrency bugs do not fall into data races [10]. To detect non-race bugs, we extend the general notion of *uncontrolled memory accesses* to conflicting accesses that are protected by critical sections, and consider them as a potential bug if the critical sections do not have an explicit communication between them.

This notion of *uncontrolled memory accesses*, which includes accesses from *uncontrolled critical sections*, provides a new condition for potential concurrency bugs. In practice, we found that this new condition is more general than data races and can detect a broad range of concurrency bugs without additional false positives. In our experiments, our detection scheme based on uncontrolled memory accesses detected all 18 real-world concurrency bugs that we tested, including atomicity violation, ordering violation, and multi-variable bugs, whereas the traditional data races could only detect 12 out of 18. Moreover, experiments on Apache, MySQL, SPLASH2, and PARSEC suggest that this new scheme does not introduce more false positives compared to traditional data race detection. The results indicate that mutual exclusion alone without additional communications, in most cases, is not sufficient for programmers to truly coordinate conflicting memory accesses.

To realize detection based on uncontrolled accesses in practice, the paper presents an efficient algorithm that can detect both data races and uncontrolled critical sections at run-time. The algorithm is based on vector clocks [6, 27], which are often used for data race detection, with extensions that consider more interleaving patterns to detect uncontrolled critical sections. While requiring larger vector clocks for mutex and additional bookkeeping for critical sections, we found that the additional meta-data is not a major concern because they are maintained only for a relatively small number of objects. The new algorithm also has a computational complexity comparable to traditional vector clock algorithms.

While effective in accurately detecting uncontrolled accesses, due to their overheads, today's vector clock schemes are often limited to testing tools instead of continuous detection on deployed systems. Unfortunately, because the size of vector clocks scale with the number of threads, efficient and scalable hardware support is quite challenging. For example, a direct hardware implementation

[23] suffers from overheads of vector clocks for each cache block and a limited number of threads that it can support. On the other hand, scalar clocks have been shown to noticeably reduce the bugs detection coverage [22].

This paper proposes architectural optimizations that address challenges in supporting vector clocks, and shows that hardware support can realize bug detection techniques based on vector clocks with negligible performance overheads without noticeably sacrificing the detection capability and scalability. The main optimization comes from the observation that only a small fraction of memory locations are accessed by multiple threads within a relatively short period where most concurrency bugs happen. As a result, we found that only storing meta-data for those shared locations can greatly reduce the overheads. Additionally, we also found that scalar timestamps can replace vector clocks in common bookkeeping with almost no loss in detection capability as long as vector clocks are maintained for critical ordering constraints. This idea is similar to a recent software optimization (FastTrack) [7] and improves scalability by allowing hardware structures to only keep scalar timestamps. Experimental results show that scalar timestamps and limited bookkeeping in hardware do not significantly impact the bug detection capability. Overall, the hardware support only requires minor architecture changes along with a small amount of state - a 6-14KB buffer per core and a 1-bit tag per data cache block.

The following summarizes the main contributions of this paper:

- *A notion of uncontrolled critical sections*: We introduce a new concurrency bug condition that can serve as a general indication of non-race concurrency bugs. The condition can be added to the traditional concept of data races to detect both data races and non-race bugs.
- *Run-time detection algorithms*: We show how vector clocks can be extended to efficiently check both data races and uncontrolled critical sections at run-time.
- *Efficient architecture support*: We show that decoupling of detecting memory locations with conflicting accesses and the rest of meta-data bookkeeping can significantly simplify hardware support with minimal impacts on coverage.

The rest of the paper is organized as follows. Section 2 presents the idea of detecting concurrency bugs based on uncontrolled accesses, which include non-race bugs. Then, Section 3 and Section 4 describe how this idea can be realized as run-time detection algorithms and architectural mechanisms, respectively. Section 5 evaluates both software and hardware implementations in terms of their effectiveness and overheads. Section 6 discusses related work, and Section 7 concludes the paper.

2 Bug Detection through Uncontrolled Critical Sections

The proposed bug detection scheme uses the notion of uncontrolled critical sections and data races to identify potential concurrency bugs. This section discusses assumptions and intuitions behind this approach, and through examples shows how the approach can detect bugs beyond traditional data races.

2.1 Assumptions

Our proposed approach shares a couple of assumptions that are common across many data race detection schemes, namely shared memory programming model and identification of synchronization

operations.

This work considers the shared memory programming model. Except for creating a thread and waiting for a termination, threads communicate through accesses to shared memory locations. Therefore, ordering of shared memory accesses characterizes communications among threads. In other words, a concurrency bug manifests as an unintended divergence in shared memory access orders in multiple program runs.

To distinguish concurrency bugs from legitimate synchronization operations, which often use races, we assume that synchronization operations can be explicitly identified. Programmers often rely on a library such as Pthreads to implement synchronization operations. In such cases, synchronization operations can be easily identified from the library calls. Our prototype implementation detects synchronization in this way. If a programmer uses custom synchronization primitives, our approach assumes that such primitives can be either marked explicitly by the programmer or automatically identified. For example, previous studies show that primitives such as spinlocks can be automatically detected [28, 26].

In addition to identifying synchronizations, our approach also distinguishes two different types of synchronization primitives. Primitives such as barriers and spinlocks explicitly enforce a predetermined ordering among threads. Therefore, the outcome of these synchronization operations are deterministic. In the discussion, we will refer to these primitives as *ordering synchronization operations*. On the other hand, primitives such as mutex and semaphores restrict the number of threads that can simultaneously execute a piece of code, often called a *critical section*, without enforcing a particular execution order. Thus, such critical sections can execute in a non-deterministic order in each run. We will refer to such primitives as *mutex synchronization*.

2.2 Concurrency Bug Detection

Parallel programs typically allow multiple threads to execute concurrently with many possible interleaving patterns. Normally, however, all interleaving patterns that are allowed by synchronization operations should produce identical program outcomes. If not, a program can produce different results for identical inputs. Informally, we consider concurrency bugs as mistakes in synchronization that allow an unintended thread interleaving pattern, which results in inconsistent program outcomes from run to run.

We note that programmers, in rare cases, may intentionally write code that produces non-deterministic results. For example, certain program outcomes such as statistics counters may not need to be precise and a programmer may choose to optimize performance by removing synchronization operations. Fortunately, our experiments on a set of benchmarks and real-world programs show that such non-deterministic outputs are rather infrequent. We assume that a programmer with knowledge of high-level semantics will review and explicitly distinguish intentional non-determinism from real bugs.

In theory, any concurrency bug can be detected if program outputs can be compared for all possible thread interleaving patterns for all possible inputs, checking for a divergence in outputs for identical inputs. Unfortunately, such exhaustive testing is infeasible in practice. Instead, our approach checks for common bug patterns where a programmer does not properly control shared memory accesses, which will lead to non-deterministic memory accesses and also possibly non-deterministic program outputs. This approach avoids multiple program executions, but requires a careful balance between false positives and negatives because non-determinism in memory accesses does not always result in a non-deterministic program output. In the following subsections,

Thread 1	Thread 2 and 3
1.1 data = compute();	2.1 Lock (l);
1.2 Lock (l);	2.2 if (**head!= tail**) {
1.3 if (**tail->next != head**) {	2.3 deQueue(data, head);
1.4 enQueue(data, tail);	2.4 **head= head->next;**
1.5 **tail = tail->next**; }	2.5 }
1.6 UnLock (*l);*	2.6 UnLock (l);

(b) A simple producer-consumer example.

Figure 1: A data race concurrency bug and a producer-consumer example

we first discuss a traditional bug condition, namely data races, and extend the intuition to a new condition, named uncontrolled critical sections.

2.3 Data Races

A large class of concurrency bugs rise from missing synchronization operations. In such cases, the bugs manifest as data races where two conflicting memory accesses execute without synchronization. Here, we define conflicting accesses as ones from different threads to the same memory location, which include at least one write. The data races imply that conflicting memory accesses can be ordered in an arbitrary fashion, and often indicate non-determinism in a program output.

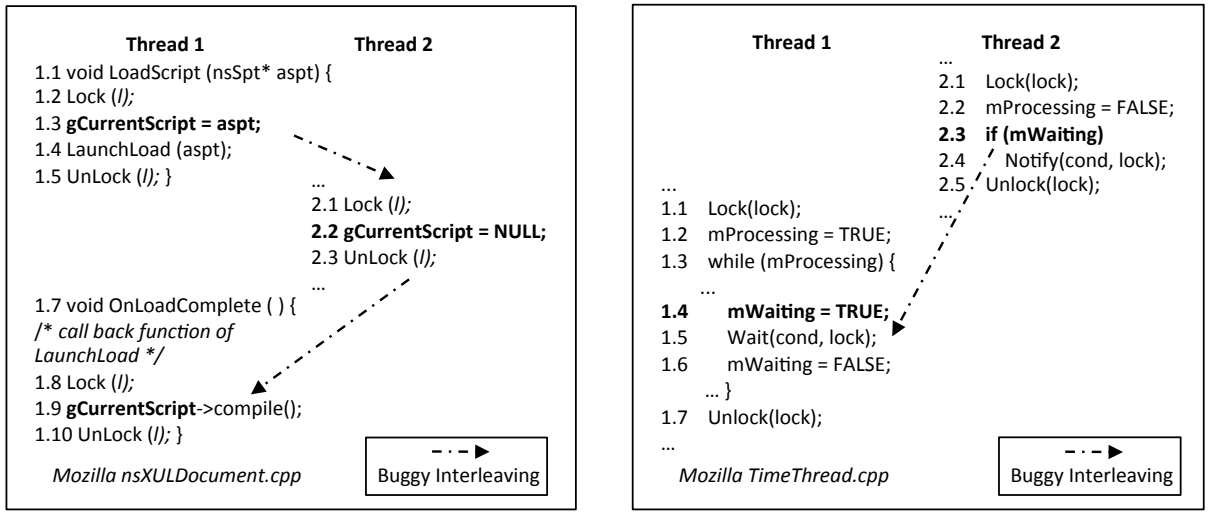
As an example, Figure 1(a) shows an atomicity violation bug in MySQL that falls into a data race. In this example, none of the accesses to the shared pointer `thd->proc_info` is protected by synchronization. As a result, these accesses can be freely reordered, and potentially resulting in a fault if the pointer is set to be NULL by *2.1* between *1.1* and *1.2*. To fix the bug, both *1.1* and *1.2* need to be protected by a single critical section to ensure an atomic execution.

Figure 1(b) shows a simple producer-consumer example where non-deterministic memory accesses do not impact the output. In the example, one thread (Thread 1) produces data and places them into a FIFO buffer so that other threads (Thread 2 and 3) can read them. The enqueue and dequeue processes are performed atomically within critical sections. Note that an order in which consumers read the FIFO does not change program outputs if both consumers perform identical operations. Even though critical sections can be executed in an arbitrary order, they are not considered as data races because of locks.

2.4 Uncontrolled Critical Sections: Non-Race Bugs

Data race detection is effective for a broad range of concurrency bugs where conflicting memory accesses are not controlled at all. However, data races cannot capture incorrect use of critical sections. For example, Figure 2(a) shows an atomicity violation example from Mozilla [11]. In this case, each memory access to the shared variable `gCurrentScript` is protected by a lock. As a result, there is no data race. However, the program may still crash when the thread interleaving follows *1.3 - 2.2 - 1.9* because `gCurrentScript` will be NULL for *1.9*. As another example, Figure 2(b) shows an ordering violation example from Mozilla. Again, there is no data race. However, the program will hang if the thread interleaving follows *2.3 - 1.4* because `mWaiting` would be `false` at *2.3* and Thread 1 waits at *1.5* for a notification that will never be sent.

In order to detect non-race bugs, we introduce a notion of *uncontrolled critical sections (UCS)*, namely two critical sections that contain conflicting accesses, yet with no explicit communication between them. In the shared memory model, an explicit communication implies a read-after-write



(a) Atomicity violation bug (KB11) [11].

(b) Ordering violation bug (KB8) [30].

Figure 2: Atomicity violation and ordering violation bugs in Mozilla that cannot be detected from data races.

(RAW) dependency between two critical sections; the later critical section reads from a memory location that is written by the earlier critical section. This condition is based on the intuition that programmers still need to coordinate critical sections in order to produce deterministic outputs even when they allow an execution order to be determined dynamically. For example, in the producer-consumer example (Figure 1(b)), a consumer needs to update the head pointer for a FIFO so that other consumers do not process the same entry many times. Because there exist RAW dependencies, the critical sections in the produce-consumer example are NOT considered *uncontrolled*. Therefore, uncontrolled critical sections do not incur a false positive in this case. In fact, this notion of uncontrolled critical sections did not show any false positive even for experiments on a broader range of benchmarks and real-world applications.

On the other hand, the uncontrolled critical sections can capture non-race concurrency bugs where critical sections are not properly coordinated. As an example, consider the example in Figure 2(a) when the program execution follows the sequence *1.3 - 2.2 - 1.9*. Even though all three accesses are protected by critical sections, the critical sections that include *1.3* and *2.2* only have a Write-After-Write (WAW) dependency and thus will be considered *uncontrolled*. Similarly, consider the example in Figure 2(b) when the thread interleaving follows *2.3 - 1.4*. In this case, the critical sections in the two threads only have a WAR dependency. Therefore, both non-race bugs can be detected with uncontrolled critical sections. We note that the ordering violation in Figure 2(b) is not detected under the correct interleaving of *1.4 - 2.3* because there exists a RAW dependency.

3 Run-Time Detection Algorithms

Here, we discuss how run-time checks for data races and uncontrolled critical sections can be realized as software algorithms. We first describe an optimized algorithm for data races, named `Race-Opt`, as a baseline and extend the algorithm to include uncontrolled critical sections, named `Race+UCS`.

detect_shared(TID, Addr, Type, TimeStamp)

1.Update the history for the memory location
  (a) If (`Type` == `Read`),
    `read_table(Addr) = (TID, TimeStamp)`.
  (b) Otherwise, `write_table(Addr) = (TID, TimeStamp)`.

2. Check the most recent write
  (a) Read `(PrevTID, PrevTimeStamp)` from `write_table(Addr)`.
  (b) If `PrevTID` $\neq$ `TID`, call `check_reorder(TID, PrevTID, PrevTimeStamp)`.

3. Check the most recent read
  (a) If (`Type` == `Read`), skip Step 3.
  (b) Read `(PrevTID, PrevTimeStamp)` from `read_table(Addr)`.
  (c) If `PrevTID` $\neq$ `TID`, call `check_reorder(TID, PrevTID, PrevTimeStamp)`.

Figure 3: An algorithm to detect conflicting memory accesses from different threads and initiate checks.

3.1 Overview

At a high-level, our run-time detection algorithms extend the general approach of the data race detection schemes that use scalar timestamps and vector clocks [27, 7]. More specifically, the algorithm first detects conflicting memory accesses, i.e. read-after-write, write-after-read, and write-after-write from two threads to the same memory location. Then, the algorithm determines if the conflicting access pair indicates either data races or uncontrolled critical sections by checking whether the accesses can be re-ordered while maintaining certain happens-before relationships [8]. For example, data races can be checked by maintaining the happens-before relationships among *all* synchronization operations so that re-ordering is only possible when there is no synchronization between the accesses, either direct or indirect. To detect both races and uncontrolled critical sections, the happens-before constraints are relaxed to only preserve the ordering between critical sections when they have a RAW dependency. The vector clocks are used to encode the happens-before relationships.

To be general, we describe our algorithms using *release* and *acquire* instead of individual synchronization operations. While there exist many types of synchronization primitives, they can fundamentally be considered as acquiring and releasing tokens. For example, mutual exclusion requires for each thread to acquire a token (lock) before entering a critical section and releases a token after the critical section. Similarly, the barrier synchronization can be realized by having each thread release its token after reaching a barrier and wait for acquiring tokens from all other threads before proceeding. For `Race+UCS`, however, the algorithm distinguishes two types of synchronization primitives: *mutex* primitives that support critical sections, and *ordering* primitives such as barriers (See Section 2.1). The algorithms also refer to synchronization tokens as *synchronization objects*.

3.2 Detection of Conflicting Accesses

The first step in our detection algorithms is to identify a pair of conflicting memory accesses: two accesses from different threads to one location with at least one being a write. The conflicting accesses represent a pair, which can lead to a change in the value read from memory if re-ordered. As an option, our algorithms can also check memory values before and after the conflicting write to see if a re-ordering will change the value. We discuss the impact of checking memory values on bug detection capability in the evaluation. However, for simplicity, this check is not shown in the algorithm here.

Figure 3 shows the operations that need to be performed on each memory access in order

to detect conflicting accesses and provide information for re-order checks. The algorithm uses read/write history tables to keep a thread ID (`TID`) and a timestamp for the most recent read and write to each memory location. This information needs to be kept at the smallest granularity that can be shared among threads; in theory, each byte. The algorithm can detect conflicting memory accesses using the thread IDs and access types (read/write) of the previous and current accesses to each location.

Upon detecting conflicting accesses, the algorithm calls the `check_reorder()` function along with information on the two accesses, in order to check if they can be re-ordered under the happens-before constraints. To aid this check, each thread maintains a local clock that increments on each synchronization operation, either acquire or release, within the thread. This clock is recorded on each memory access as a timestamp to indicate when the previous access to each location happened. This clock is one of the thread vector clocks (`ThreadVClk[TID][TID]`) that we discuss in the following subsection.

Rather than being comprehensive, our algorithm only detects conflicts with the most recent read and write to each memory locations. Traditional vector clocks typically keep track of the most recent read and write from each thread to each location [6, 27, 23]. However, a recent study showed that keeping one write per location instead of a vector of writes from each thread is sufficient to provide a comprehensive coverage for data races [7]. We take this optimization approach even further by keeping only one write and one read per location. In practice, we found that detecting conflicts with the most recent read and write is sufficient to detect almost all concurrency bugs or even injected races (see Section 5).

3.3 Checks for Memory Access Reordering

Once a conflicting memory access pair is detected, the algorithm checks if the accesses can be re-ordered within the imposed happens-before constraints. For this purpose, our algorithms use vector clocks [6]. For a parallel program with `N` threads, each thread maintains a vector clock with `N` elements. `ThreadVClk[i][i]` represents Thread `i`'s local clock that is incremented on each synchronization operation within that thread. Conceptually, other elements in a vector clock encode the ordering constraint between two threads. For example, `ThreadVClk[i][j]` indicates the earliest that the current memory access from Thread `i` can be moved to in terms of Thread `j`'s local time. If the vector clocks are properly maintained, one can check if the current memory access from Thread `i` can be reordered before a previous access from Thread `j` simply by comparing Thread `i`'s vector clock value `ThreadVClk[i][j]` with the timestamp of the previous access.

3.3.1 Race Detection (`Race-Opt`)

Conflicting accesses represent a data race if they can be re-ordered while maintaining the happens-before relationships among *all* synchronization operations. In this case, maintaining vector clocks is straightforward because every synchronization operation adds a re-ordering constraint in the same way. Figure 4 shows the algorithm. To encode the ordering constraints from each synchronization operation, the algorithm maintains a vector clock for each synchronization object (object VC) in addition to thread vector clocks (thread VCs). On a release operation, the object VC is updated with the vector clock for the thread that performs the release (take the later timestamp for each element). The object VC represents the earliest that the following release operation can

detect_shared(TID, Addr, Type, TimeStamp)

Meta-data:
1.`ThreadVClk[TID1][TID2]`: A vector clock per thread.
(`N` elements for a program with `N` threads).
2. `ObjVClk[SyncObj][TID]`:
A vector clock per synchronization object.

Functions:
`all_update_release(TID, SyncObj)`
1.For each element in the vector clock, `ObjVClk[TID][i]` =
`MAX(ThreadVClk[TID][i], ObjVClk[SyncObj][i])`.

Functions:
`all_update_acquire(TID, SyncObj)`
1. For each element in the vector clock, `ThreadVClk[TID][i]` =
`MAX(ThreadVClk[TID][i], ObjVClk[SyncObj][i])`.

`all_check_reorder(TID, PrevTID, PrevTimeStamp)`
1.Check if the memory accesses can be re-orderd:
If `ThreadVClk[TID][PrevTID]` $\leq$ `PrevTimeStamp`,
report a data race.

Figure 4: An algorithm to determine if two memory accesses represent a data race.

detect_shared(TID, Addr, Type, TimeStamp)

Meta-data:
1. `ThreadVClk[TID1][TID2]`: A vector clock for each thread.
2. `OrderObjVClk[SyncObj][TID]`:
A vector clock for each ordering synchronization object.
3. `MutexObjVClk[SyncObj][TID1][TID2]`:
A set of vector clocks (one per thread)
for each mutex object.
4. `CSList[TID]`:
A list of critical section(s) that a thread is currently in.
5. `DetectionList[TID]`:
A list of potentially buggy memory access pairs.

Changes to
`detect_shared(TID, Addr, Type, TimeStamp)`:
1. Include `CSList[TID]` in `read_table()`
and `write_table()`;
(a) Store `(TID, TimeStamp, CSList[TID])`
in each memory access.
(b) Use `PrevCSList` from the table
when calling `check_reorder()`.
2. Call `raw_update_raw()` when a Read-After-Write is detected.
`WrCSList` is `CSList` from the `write_table()`.

Functions:
`raw_update_acquire(TID, SyncObj)`
1. If (`type(SyncObj)` == `OrderSyncObj`),
for each `i`, update the vector clock: `ThreadVClk[TID][i]` =
`MAX(ThreadVClk[TID][i], OrderObjVClk[SyncObj][i])`.
2. If (`type(SyncObj)` == `MutexSyncObj`),
add `SyncObj` to `CSList[TID]`.

Functions:
`raw_update_release(TID, SyncObj)`
1. If (`type(SyncObj)` == `OrderSyncObj`),
For each element in the vector clock,
`OrderObjVClk[TID][i]` = `MAX(ThreadVClk[TID][i],`
`OrderObjVClk[SyncObj][i])`.
2. If (`type(SyncObj)` == `MutexSyncObj`),
(a) `MutexObjVClk[SyncObj][TID]` = `ThreadVClk[TID]`.
(b) Remove `SyncObj` from `CSList[TID]`.
(c) If (`CSList[TID]` == `NULL`),
for each access pair in `DetectionList[TID]`,
if `ThreadVClk[TID][PrevTID]` $\leq$ `PrevTimeStamp`
report a bug.

`raw_update_raw(RdTID, WrTID, WrCSList)`
1. If ((`CSList[RdTID]` $\cap$ `WrCSList`) $\neq$ `NULL`),
for each element in (`CSList[RdTID]` $\cap$ `WrCSList`),
update the thread vector clock: `ThreadVClk[TID][i]` =
`MAX(ThreadVClk[RdTID][i],`
`MutexObjVClk[SyncObj][WrTID][i])`.

`raw_check_reorder(TID, PrevTID, PrevTimeStamp, PrevCSList)`
1. If (`ThreadVClk[TID][PrevTID]` $\leq$ `PrevTimeStamp`),
(a) If (`CSList[TID]` $\cap$ `PrevCSList` $\neq$ `NULL`)
Add an entry to `DetectionList[TID]`.
(b) Otherwise, report a bug detection.

Figure 5: An algorithm to determine if two memory accesses represent an uncontrolled critical section.

happen in each thread's local time. On an acquire operation, a thread VC is updated with the corresponding object VC.

3.3.2 `Race+UCS`

Figure 5 shows how the vector clocks can be managed for the `Race+UCS` scheme. For the ordering synchronization, the algorithm is identical to `Race-Opt` because both enforce the same happens-before constraints. However, for the mutex synchronization, the `Race+UCS` algorithm is more complex because an ordering between two critical sections is maintained only when there

exists a read-after-write dependency. For this purpose, the algorithm adds additional structures to selectively update object vector clocks, detect a RAW dependency between critical sections, and delay a detection.

In the `Race+UCS` algorithm, a vector clock needs to be propagated from one thread to another through a mutex object only for a subset of release-acquire pairs. In order to support such a behavior, our algorithm keeps multiple vector clocks for each mutex object one from each thread, (`MutexObjVClk[SyncObj][TID1][TID2]`). On a release operation, a thread updates its own mutex object vector clock. In this way, a thread can inherit the correct vector clock from the critical section with a RAW dependency even when it is not the most recent release operation for the mutex object.

To detect a RAW dependency between two critical sections, the algorithm keeps track of which critical sections that each thread is currently in (`CSList[TID]`) and was in for memory operations recorded in the read/write history tables. Upon detecting a RAW dependency between two memory accesses, the lists allow the algorithm to check (in `raw_update_raw()`) if the two accesses are both inside critical sections that are protected by the same mutex object. If so, the thread vector clock is updated with the corresponding mutex vector clock. This dynamic approach introduces delays in both propagating vector clocks and reporting a bug because the dependency between critical sections is not known at the beginning of a critical section. Therefore, when a potential bug is detected within a critical section, the bug is recorded in a list (`DetectionList`) and the report is delayed till the end of the critical section when an updated thread vector clock is available.

The computation complexity of the `Race+UCS` algorithm is comparable to that of `Race-Opt` because the only major difference is in a constant overhead of accessing `CSList`. Our Pin-based implementations also show comparable performance for both algorithms, and confirm this observation. The main overheads of `Race+UCS` compared to `Race-Opt` comes from the storage for mutex object vector clocks. `Race-Opt` requires $N_{mutex} \times N_{thread} \times B_{ts}$ where N_{mutex} is the number of mutex objects, N_{thread} is the number of threads, and B_{ts} is the size of a timestamp. On the other hand, `Race+UCS` requires $N_{mutex} \times N_{thread} \times N_{thread} \times B_{ts}$, which grows much faster with the number of threads. However, we found that the storage overhead is reasonable under realistic parameters. For example, even with 100 threads and 1,000 mutex objects, the storage requirement is 20MB for 16-bit timestamps. We also believe that the algorithm can be further optimized as the mutex vector clocks are very sparsely utilized.

4 Architecture Support

This section describes the hardware support for an efficient realization of the proposed run-time detection schemes, enabling them to be applied to production systems.

4.1 Challenges

The main challenge in hardware support for concurrency bug detection lies in managing meta-data efficiently without significantly sacrificing scalability or detection coverage. A large amount of meta-data could result in large hardware structures or noticeable interference with regular program execution. On the other hand, reducing the amount of meta-data may limit the maximum number of threads that hardware can support or result in undetected bugs. In this context, traditional vector clocks [27, 23] are particularly challenging to support in hardware because they require a vector

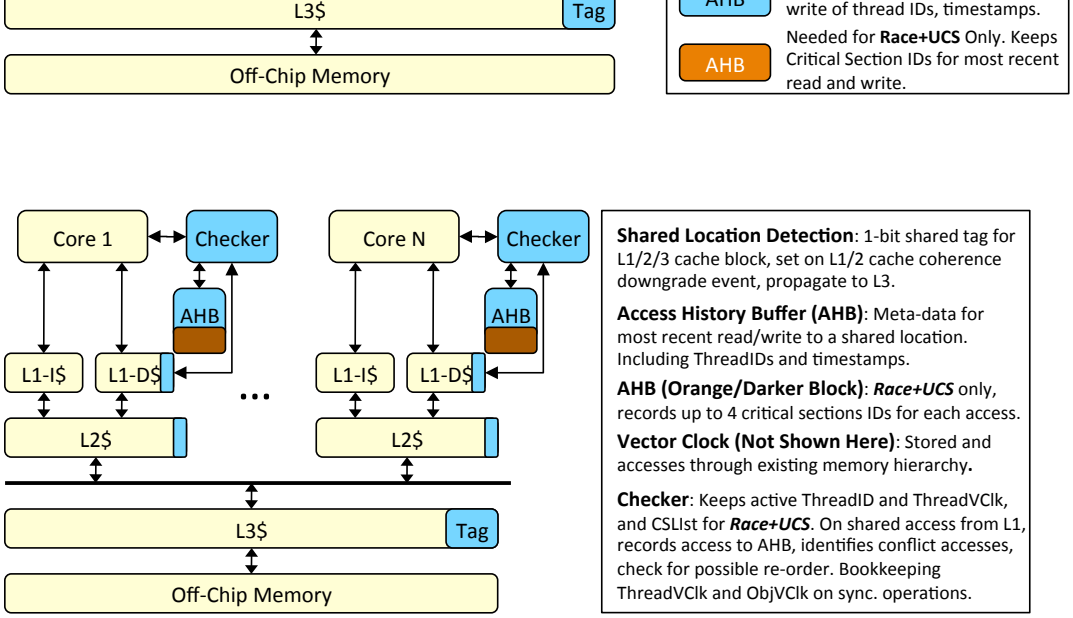


Figure 6: A block diagram for the overall architecture. The dark blocks represent new additions for concurrency bug detection; blue blocks are common for both `Race-Opt` and `Race+UCS`; orange blocks represent additional space only for `Race+UCS`.

clock for each memory location, whose size increases linearly with the number of threads. Even if we keep scalar timestamps for most memory locations as in our schemes, the two timestamps and thread IDs per word (or byte) represent significant overheads (2.5x for 16-bit timestamps and 8-bit thread IDs).

In order to manage the overheads, previous proposals for happens-before data race detection in hardware store meta-data at a coarse granularity, often one or two for each cache block [23, 22]. Also, these designs integrate the meta-data into data caches, introducing additional storage for each cache block. Unfortunately, such integrated designs trade-off flexibility and accuracy for lower overheads. For example, ReEnact [23] can only support up to 4 threads and detect one conflicting access for each cache block. CORD [22] uses scalar timestamps in place of vector clocks, but can detect only about 70% of data races compared to vector clocks. Ideally, the hardware support should have low overheads while allowing fine-grained bookkeeping to maintain high accuracy and detection coverage.

4.2 Approach: Decoupled Bookkeeping

The architecture design in this section uses the intuition that most of bookkeeping for concurrency bug detection is only necessary for "shared" memory locations. For example, our algorithms use scalar timestamps and vector clocks only if conflicting accesses are detected. Here, we use the term "shared" to refer to locations with conflicting accesses. Moreover, such shared memory locations are a small fraction of the entire memory space, especially for a relatively small window where most conflicting accesses happen. In our study on PARSEC and SPLASH2 benchmarks, less than 1% of memory locations have conflicting accesses that happen less than 100,000 memory accesses apart. Therefore, keeping meta-data such as timestamps and vector clocks for all memory locations, even just for on-chip caches, is extremely wasteful. Instead, in our design, we decouple detection of conflicting accesses and the rest of bookkeeping so that most meta-data are stored only for memory locations with conflicting accesses.

4.3 Architecture Design

Figure 6 shows the high-level block diagram for our architecture with support for concurrency bug detection. In the figure, the blue blocks indicate the new hardware components that are common

for both `Race-Opt` and `Race+UCS`, and the orange blocks show additional bookkeeping only for `Race+UCS`. The high-level architecture is virtually identical for both detection schemes except `Race+UCS` maintains additional state (in AHB) and performs more complex operations (see Section 3).

4.3.1 Extension for Shared Location Detection

Our architecture combines a simple cache tag and a coherence protocol to detect shared memory locations that were recently accessed by multiple threads, with at least one write. The rest of the bookkeeping is performed only for those locations that are marked as *shared* by this mechanism. More specifically, we augment each block in data caches with a 1-bit tag, which indicates whether the block is shared or not. This bit is cleared when a cache block is evicted or read from off-chip memory. The shared bit is set when a cache coherence event indicates that multiple cores access the same cache block with at least one write. For example, a request from a remote cache to have an exclusive copy when there exists a local copy indicates the block is shared. This tag bit follows the data on-chip; a dirty tag is written back to the lower level, and the tag is read with data on a cache miss. Effectively, this mechanism detects memory blocks that are shared within a window, while the block exists in multiple private (L1/L2) caches, and keeps this history while the block is on-chip.

We believe that the 1-bit tag is sufficient under the assumption that each core runs one thread with infrequent context switches or thread migrations. If not, the 1-bit tag may not detect locations that are shared by multiple threads on one core or incorrectly identify a block as shared when a single thread moves from one core to another. However, note that the inaccuracy can only lead to false negatives, but not false positives. To be more accurate, a thread ID can be added to the 1-bit tag in order to identify where each access comes from. The overhead will be still quite low even with the thread ID as only one ID needs to be maintained for a cache block.

4.3.2 Checker Module

A checker module for each core maintains per-thread state and performs most of the bookkeeping and checking operations. For the active thread on the core, the checker keeps a thread ID, a thread vector clock, and a list of current critical sections for `Race+UCS`. On a memory access from the core, the checker module uses the L1 data cache tag to determine if the access is to a "shared" block. If so, the checker records the access into the access history buffer (AHB), identifies for conflicting accesses, and checks if a re-ordering is possible. The checker module also coordinates with software layers through new instructions. For `Race-Opt`, the architecture provides two additional instructions to indicate a synchronization operation, one for *acquire* and the other for *release*. For `Race+UCS`, the architecture adds four instructions, distinguishing ordering and mutex synchronization operations. These instructions also convey the address of the synchronization object vector clock(s).

4.3.3 Access History Buffer (AHB)

The access history buffer (AHB) records information on the most recent read and write to a shared location that can be used to detect conflicting accesses and to check re-ordering. Essentially,

the AHB serves as the read and write history tables in our algorithms (see Section 3.2). On a shared memory access, the checker module records a thread ID and a timestamp into the AHB. For `Race+UCS`, the AHB also records up to four critical section IDs for each access.

As the AHB has a limited capacity, it works like a cache and only keeps the history of recently shared memory accesses. However, there is no backup hierarchy for the AHB. If an entry is evicted from an AHB, the information is simply thrown away. A miss to AHB creates a new entry. While this design implies that we cannot detect conflicting accesses with a long distance in between, our experimental results suggest that an AHB with 1024 entries, which correspond to 6-KB for `Race-Opt` and 14-KB for `Race+UCS` assuming 8-bit thread IDs and 16-bit timestamps, are sufficient for virtually all bugs tested.

We note that the AHB can store an access history *per word* because only accesses to shared memory locations are recorded. On the other hand, traditional designs that combine meta-data into the main cache often had to store information on a cache block granularity to keep overheads acceptable.

4.3.4 Vector Clocks

Our algorithms use two types of vector clocks: thread vector clocks and synchronization object vector clocks. For the thread vector clock, our architecture uses dedicated storage in each checker module for an active thread on the core. We found that the thread vector clocks need to be close to the checkers because they are used in each re-order check operation. The thread vector clock needs to be treated as a part of thread state and managed by an operating system on a context switch. On the other hand, vector clocks for synchronization objects are stored and accessed through the existing memory hierarchy. For each synchronization object, software allocates space for a vector clock in its memory space and passes the location using the new instructions that indicate synchronization operations.

Hardware counters have a limited number of bits. As a result, the clock that each thread uses to represent its local time may overflow after many synchronization events. Fortunately, our experiments show that synchronization operations are rather infrequent and the thread clocks only increment slowly. In fact, we did not see any overflow for PARSEC and SPLASH2 benchmarks even with 16-bit counters. Given that overflows are infrequent, our architecture handles them in a relatively slow but straightforward fashion instead of adding complex hardware. Upon detecting an overflow in its local clock, a checker raises an exception to an operating system, which in turn interrupts other cores that run other threads from the same program. Then, the operating system resets all timestamps and vector clocks to zero, and sets each thread's local time to one to prevent any false positives. In order to allow an operating system to clear vector clocks for synchronization objects, an application allocates them in separate pages that are known to the operating system.

5 Evaluation

5.1 Evaluation Setup

Our infrastructure is built on the Pin binary instrumentation framework [15]. To evaluate the detection capabilities, our tool implements the bug detection algorithms by intercepting memory accesses and pthread calls. To evaluate the architectural support, we implemented a typical memory hierarchy with bookkeeping structures in a Pin tool, and also added a timing model with a

Table 1: Baseline architecture parameters.

Component	Parameters
Core	4 2-GHz in-order single-issue cores
Caches	L1 I/D (private): 32KB/32KB 4-ways 3 cycles Latency L2 (private): 256KB, 4-ways 15 cycles latency L3 (shared): 8MB, 8-ways 40 cycles latency
Coh. protocol	MSI
DRAM	4GB 50ns Latency
Meta-data	8-bit thread/lock ID, 16-bit clock
AHB	1024-entries, 1-way, 6KB (Race-Opt) / 14KB (Race+UCS)

Table 2: Detection capabilities (A - Atomicity violation, O - Order violation, M - Multi-variable bug).

	Race-VC SW Only	Race-Opt SW	Race-Opt HW	Race+UCS SW	Race+UCS HW
Apache-A (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
MySQL-A (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
MySQL-A (Non-Race)	No	No	No	**Yes**	**Yes**
KB1-A(MySQL) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB2-AM(MySQL) (Non-Race)	No	No	No	**Yes**	**Yes**
KB3-A(Apache) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB4-A(MySQL) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB5-O(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB6-O(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB7-O(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB8-O(Mozilla) (Non-Race)	No	No	No	**Yes***	**Yes***
KB9-O(MySQL) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB10-AM(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB11-AM(Mozilla) (Non-Race)	No	No	No	**Yes**	**Yes**
KB12-AM(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB13-AM(Mozilla) (Race)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
KB14-A(MySQL) (Non-Race)	No	No	No	**Yes***	**Yes***
KB15-A(MySQL) (Non-Race)	No	No	No	**Yes**	**Yes**

processing core that runs 1 instruction per cycle, L1/L2/L3 caches, and a memory interface.

Table 1 summarizes the baseline architecture parameters. We model a multi-core processor with 4 cores, 64KB L1 and 256KB L2 private caches per core, and an 8MB shared L3 cache. We also model MSI cache coherence protocol. For the additional bookkeeping, we model the access history buffer (AHB) with an 8-bit thread ID and a 16-bit timestamp per access. For `Race+UCS`, the AHB also includes up to four 8-bit lock IDs per access to record critical sections. Each entry records one read and one write.

To evaluate the bug detection capability, we use two types of benchmarks, namely kernel bugs (KB) and real programs. The kernel bugs are created based on real-world application bugs (MySQL, Apache, and Mozilla) from previous studies [11, 10, 30]. The kernel bugs use 2 threads to reproduce the original bugs. We also use three concurrency bugs from two large real-world server applications (Apache and MySQL). We use 30 threads for Apache and 10 threads for MySQL, respectively. For a further study on coverage, we also perform random race injections to benchmarks from SPLASH2 [3] and PARSEC [1]. The race injection is performed by randomly selecting a lock/unlock pair and ignoring the pair for the entire program execution. The SPLASH2 and PARSEC benchmarks are run using 4 threads with the default input size for SPLASH2 benchmarks and `simmedium` input size for PARSEC.

5.2 Bug Detection Capability

We compare the bug detection capabilities of three schemes: `Race-VC`, `Race-Opt`, and `Race+UCS`. `Race-VC` represents a traditional vector clock scheme [6, 27] where a vector clock is kept for each

Table 3: Race injection study results. 50 races were randomly injected into the benchmarks to measure the detection capability. (P) - PARSEC, (S) - SPLASH2.

	Race-VC	Race-Opt/Race+UCS (SW)	Race-Opt (HW)	Race+UCS (HW)
Blackscholes (P)	50	50	50	50
Bodytrack (P)	50	50	50	50
Fluidanimate (P)	50	50	49	50
LU (S)	50	50	50	50
Ocean (S)	50	49	49	48
Radiosity (S)	50	50	50	50
Radix (S)	50	50	50	50
Swaptions (P)	50	50	50	50
Water-nquare (S)	50	50	49	50
Water-spacial (S)	50	50	50	50
Total	500/500	499/500	497/500	498/500

Table 4: The number of detections without a bug.

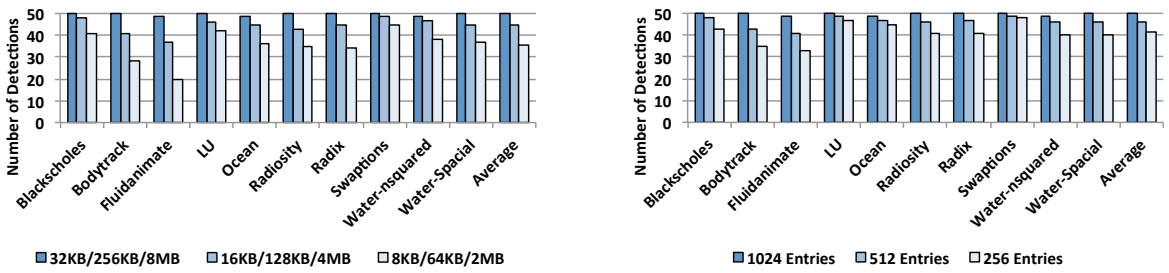
	Race-VC (SW)	Race-Opt (SW)	Race-Opt (HW)	Race+UCS (SW)	Race+UCS(HW)
Apache	9 (2 benign)	9 (2 benign)	9 (2 benign)	9 (2 benign)	9 (2 benign)
Others	0	0	0	0	0

memory location. `Race-Opt` uses scalar timestamps for the most recent read/write instead.

5.2.1 Detection Coverage

Table 2 shows detection results for real-world bugs. The results show that both `Race-Opt` and `Race-VC` detect all data-race bugs, indicating scalar timestamps do not significantly affect detection coverage. More importantly, `Race-UCS` detects all bugs including 6 non-race bugs, showing the notion of uncontrolled critical sections is indeed effective in practice. While none of these schemes is explicitly designed for multi-variable bugs, the results show that at least some multi-variable bugs can be detected by checking non-determinism in a single variable. Note `Race+UCS` detects KB8 only when the bug manifests as discussed in Sections 2.4. Similarly, KB18 is detected only on certain program runs. This is because the bug involves a write and a read in two critical sections. Our scheme detects the bug when the write happens after the read, forming a WAR dependency, but cannot detect it in the opposite case that forms a RAW dependency.

In theory, `Race-VC` may be able to detect data races that `Race-Opt` cannot. This is because the full vector clock scheme keeps the most recent read/write from each thread whereas `Race-Opt` only keeps one read and one write to detect conflicting accesses. Also, hardware implementations rely on caches to detect shared locations and the AHB to keep recent accesses. Because both caches and the AHB have limited capacities, relevant information may be evicted, causing potential false negatives. However, results for both real-world bugs in Table 2 and injected races in Table 3 suggest that our proposed algorithms have a minimal impact on coverage in practice when compared to `Race-VC`. Previous studies [10, 14] also observed that real-world concurrency bugs typically manifest within a short window and involve two threads. This observation explains why keeping the most recent read and write rather than accesses from each thread is enough in most cases. This also explains why the hardware implementations show comparable detection coverage to the software implementations.



(a) Cache size study.

(b) AHB size study.

Figure 7: The impact of caches and AHB sizes on detection capabilities of `Race-Opt`. We injected 50 races to each configuration. We reduced L1, L2, L3 and AHB sizes to 1/2 and 1/4 of the baseline configurations (32KB, 256KB, 8MB, 1024-entries).

5.2.2 False Positive Study

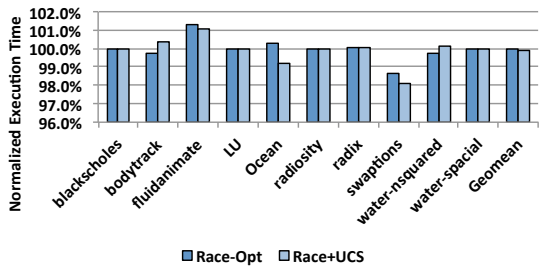
Ideally, a bug detection scheme should only report real bugs. As shown in Table 4, our detection schemes, in both software and hardware implementations, report no false positive for SPLASH2, PARSEC, and MySQL without a bug. More importantly, `Race+UCS` has the exact same false positives as race detectors, `Race-VC` and `Race-Opt`, showing that uncontrolled critical sections do not introduce new false positives.

For Apache, our schemes detected 7 intended data races that change outputs and 2 benign races that do not impact outputs. Programmers may introduce an intended race in the program if the race does not affect the correctness. For example, many of the intended races in Apache are either introduced by accesses to the time logging variables or deadlock watchdog counters. The code comments indicate that programmers chose to allow errors in logs for better performance. The watchdog counters count the number of iterations to prevent deadlocks, and do not need to be accurate.

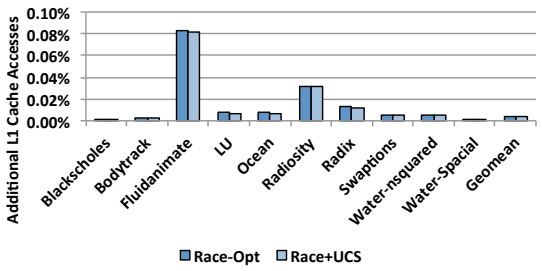
Two of the nine races in Apache are benign and do not affect the program output. Those two races happen during an initialization, where multiple threads initialize the same variables to the same value. The two benign races will not be detected if the memory values are checked.

5.2.3 Cache and AHB Size Analysis

The hardware-based scheme relies on caches and the AHB for bookkeeping. Therefore, the cache and AHB sizes directly affect the detection capability. The race injection study in Figure 7(a) shows the impact of reducing cache sizes on the detection coverage. Here, the L1, L2, and L3 caches are reduced to 1/2 and 1/4 of the baseline while keeping the AHB at the baseline size. As expected, the detection rate decreases as the cache sizes decrease. We also varied individual cache sizes (not shown) and found that both L2 and L3 cache sizes have comparable impacts. Similarly, Figure 7(b) shows the impact of reducing the AHB size. The coverage decreases as the AHB size decreases because smaller AHBs can keep history for less memory locations. The exact impact of reduced cache and AHB sizes, however, depends on application characteristics. For example, memory intensive benchmarks such as `Fluidanimate` are more sensitive than others. Overall, the experiments indicate that our scheme needs a private (L2) cache of 128-KB, the last level cache (L3) of 4-MB, and the AHB with 512 entries in order to provide good coverage (90% detection



(a) Performance overhead normalized to baseline



(b) Additional VC accesses with respect to the data accesses

Figure 8: Hardware performance study shows negligible overhead due to small number of additional vector clock (VC) accesses with respect to the data accesses. Numbers are normalized to baseline (i.e. no detection scheme) results.

rate).

5.3 Performance Overheads

Currently, both software implementations based on `Race-Opt` and `Race+UCS` incur 10-20X performance overhead compared to the native execution. The overhead includes the standard overhead from Pin. The current software implementations are not optimized, and we expect the overheads can be lower.

Figure 8(a) show the normalized execution time for the hardware-based detection schemes. The performance overheads are negligible on average. Even in the worst case, the overhead is 1.0% for `Fluidanimate`. Because the architecture mostly uses dedicated on-chip structures such as 1-bit cache tags and the AHB for bookkeeping, the only major source of performance overhead comes from accessing vector clocks for synchronization objects through the normal cache hierarchy. As shown in Figure 8(b), however, the number of vector clock accesses are negligible for both schemes when compared to the number of regular data accesses. For `Race-Opt`, the vector clocks only introduce 0.08% more accesses for `Fluidanimate` in the worst case, and 0.003% more on average. Hence, the overall performance impact due to additional vector clock accesses is negligible. While not shown here due to space constraints, the average L1 cache miss rate only increases 0.03% compared to the baseline rate for both detection schemes. In some cases, we found that the vector clock accesses may improve cache performance by changing access patterns. In `Swaptions` the L1 miss-rate is reduced by 0.1% with vector clock accesses.

In addition to vector clock accesses, counter overflows may introduce performance overheads by requiring timestamps and vector clocks to reset. However, we have never encountered any overflow during experiments. In the worst case in `Fluidanimate`, we found the maximum timestamp value of 47,832 after 287,748,471 memory accesses before the benchmark finishes. In all other benchmarks, the timestamp values never exceed 10,000 after the entire execution. Hence, we believe that the possible performance overhead introduced by timestamp overflows is negligible.

6 Related Work

6.0.1 Data Race Detection

At a high-level, data race detection techniques can be categorized into static and dynamic approaches. Static race detection schemes such as RacerX [5] use static analysis to detect possible data races. However, static approaches are generally conservative without run-time information, and usually require source code.

Dynamic data race detection techniques fall into two main classes, namely lockset based and happens-before relation based. The lockset approach such as Eraser [25] checks whether each shared variable is protected by at least one lock during the execution. The happens-before approach checks whether two memory accesses are explicitly synchronized [8]. There are many previous proposals that fall into this category, including RecPlay [24], Light64 [18], ReEnact [23], CORD [22], FastTrack [7] and others. In general, the happens-before approaches are more accurate than the lockset approaches but often have higher overheads. Researchers have also investigated hybrid approaches in order to reduce the overhead of happens-before algorithms while maintaining a low false positive rate [19, 4, 21].

The proposed bug detection techniques can be considered as an extension of the happens-before approach to detect bugs at run-time. However, unlike any previous data race detection schemes, we introduce the new notion of uncontrolled critical sections, which enables us to detect bugs beyond traditional data races. Also, our hardware architecture shows that the happens-before race detection approach as well as `Race+UCS` can be realized in hardware with minimal performance overheads and with minimal impact on coverage.

6.0.2 Concurrency Bug Detection (Beyond Data Races)

Recently, there have been significant efforts to detect concurrency bugs using symptoms beyond data races. One popular approach is to detect and/or tolerate bugs based on common program behaviors. For example, AVIO [11] and SVD [29] approximate intended atomic regions using common behaviors, Atom-Aid [14] tries to dynamically avoid atomicity violation bugs, MUVI [9] and ColorSafe [13] target to detect concurrency bugs that involve multiple variables, and Bugaboo [12] detect anomalies in communication graphs. Alternatively, researchers have also found that concurrency bugs can be identified from their consequences such as memory errors [32] or other failure patterns [31]. Traditionally, a program is often tested by running with many possible interleaving patterns and checking results [16, 20, 2]. This approach can detect any concurrency bug if a buggy interleaving is tried, yet can only test one case at a time.

This work presents a new approach to detect non-race bugs by extending the intuition behind data races to uncontrolled critical sections. This approach can be applied to programs without learning application-specific or bug-specific behaviors, and can often identify potential bugs before they happen at run-time. However, this technique is still empirical and there is no guarantee on bug detection coverage. In this sense, the proposed scheme complements an existing body of work in non-race bug detection.

6.0.3 Hardware-Based Race Detection

While many concurrency bug detection techniques can be enhanced with architecture support, this work is most closely related to happens-before race detection. In this context, ReEnact [23] provides hardware support for logical vector clocks [6] for cache lines. CORD [22] avoids the overheads of vector clocks by keeping four scalar timestamps per cache line, at the expense of smaller coverage. Both of these approaches integrated meta-data into each cache line, paying overheads on every cache line. Our work shows that decoupling of detecting shared memory locations and the rest of the bookkeeping can significantly reduce hardware and performance overheads with minimal impact on coverage. We also show that a combination of scalar timestamps and vector clocks help enable efficient implementations without sacrificing detection capabilities.

As an alternative to the happens-before approach, researchers have also presented simple hardware support for race detection relying on other heuristics. For example, HARD [33] uses locksets and SigRace [17] uses hash signatures from Bloom filters to detect possible data races. These approaches enable reasonable race detection capability with minimal hardware additions. However, generally they trade-off accuracy and coverage for the simplicity. In this work, we showed that accurate happens-before approaches can also be realized with relatively simple hardware support.

7 Conclusion

This paper introduces the notion of uncontrolled critical sections, which captures improperly coordinated critical sections, and presents a general bug detection approach that covers both data races and non-race bugs. This new approach can be realized by extending traditional happens-before race detection algorithms. Experimental results show that a broad range of concurrency bugs can be detected in this fashion without introducing false positives. In addition to the algorithms, we also show a decoupled architecture design can enable the happens-before detection schemes to be realized with minimal overheads and without sacrificing detection coverage, thus providing an attractive way to detect concurrency bugs.

References

[1] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the* 15^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, 2010.

[3] CAPSL. The modified SPLASH-2. `http://www.capsl.udel.edu/splash/`, July 2007.

[4] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[5] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the* 19^{th} *ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[6] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, August 1991.

[7] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[8] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[9] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of* 21^{st} *ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116, 2007.

[10] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[11] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the* 12^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[12] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the* 42^{nd} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 553–563, 2009.

[13] Brandon Lucia, Luis Ceze, and Karin Strauss. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the* 37^{th} *Annual International Symposium on Computer Architecture*, pages 222–233, 2010.

[14] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the* 35^{th} *Annual International Symposium on Computer Architecture*, pages 277–288, 2008.

[15] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the* 2005 *Conference on Programming Language Design and Implementation International (PLDI)*, June 2005.

[16] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the* 8^{th} *USENIX Conference on Operating Systems Design and Implementation*, pages 267–280, 2008.

[17] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture*, pages 337–348, 2009.

[18] Adrian Nistor, Darko Marinov, and Josep Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *Proceedings of the* 42^{nd} *Annual IEEE/ACM International Symposium on Microarchitecture*, pages 541–552, 2009.

[19] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the* 9^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[20] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceeding of the* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[21] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the* 9^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–190, 2003.

[22] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2006.

[23] Milos Prvulovic and Josep Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the* 30^{th} *Annual International Symposium on Computer Architecture*, pages 110–121, 2003.

[24] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:133–152, May 1999.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

[26] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 143–154, 2008.

[27] Céline Valot. Characterizing the accuracy of distributed timestamps. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 43–52, 1993.

[28] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the* 9^{th} *USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, 2010.

[29] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.

[30] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the* 36^{th} *annual International Symposium on Computer Architecture*, pages 325–336, 2009.

[31] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: detecting concurrency bugs through sequential errors. In *Proceedings of the* 16^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 251–264, 2011.

[32] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the* 15^{th} *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–192, 2010.

[33] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132, 2007.