EFFECTIVE ARCHITECTURAL SUPPORT FOR DETECTING CONCURRENCY BUGS

BY

ABDULLAH AL MUZAHID

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Marc Snir
Associate Professor Darko Marinov
Assistant Professor Sam King
Dr. Matthew Frank, Intel
Dr. Paul Petersen, Intel

Abstract

Multicore machines have become pervasive and, as a result, parallel programming has received renewed interest. Unfortunately, writing correct parallel programs is notoriously hard. Therefore, it is important to innovate with techniques and approaches to tackle various types of concurrency bugs.

This thesis aims at making parallel programming easier by detecting some of the most common and difficult concurrency bugs in shared memory parallel programs, namely data races, atomicity violations, and sequential consistency violations. Specifically, we propose novel, effective and efficient hardware-based techniques that help detect and isolate these bugs. We use hardware-based solutions because they lead to low overhead solutions. Therefore we can use these techniques to detect the bugs both during development time and during production run.

The proposal to detect data races is called SigRace. It uses hardware address signatures to detect data races dynamically at run time. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them with those of other processors. If the intersection is not null, a data race may have occurred, in which case we run a more detailed analysis to pinpoint the race.

SigRace can detect data races successfully. But even if a multithreaded program does not have any data races, it can still show incorrect behavior because of a bug named atomicity violation. The proposal to detect atomicity violations is called AtomTracker. It is based on first trying to learn atomicity constraints automatically from the program, by analyzing many correct executions. After it finds the set of possible atomic regions, the hardware monitors the execution to detect

any violations of these atomic regions. AtomTracker uses a hardware very similar to SigRace to accomplish this.

The above approaches tackle data races in a classical sense or in a more higher level sense (namely atomicity violations). The last work of this thesis is to find out a special pattern of data races that are particularly hard to detect and analyze. This complicated pattern of data races leads to violation of sequential consistency which is the underlying behavior of the memory model that programmers usually assume. Sequential consistency violations (SCV) lead to some of the most notorious bugs in parallel programs. In order to detect SCV in a machine with a relaxed memory model, we leverage cache coherence protocol transactions and dynamically detect cycles in memory-access orderings across threads. When one such cycle is about to occur, an exception is triggered, providing the exact architectural state.

We performed detailed experimentation with each of these techniques and showed that they are effective in detecting various types of concurrency bugs. More importantly, we uncovered several *new and previously unreported* bugs in various popular open source codes using these solutions.

To my loving parents, Laz, and Sadit.

Table of Contents

Chapter 1

Introduction

Advances in semiconductor industry has ushered a new era of multicore machines. Multicore machines are becoming so pervasive that it would be unwise not to take advantage of the increased parallelism that they provide. Therefore, parallel programs are becoming more important than ever before. However, parallel programming comes with its own unique set of problems. Arguably, the most important one is the concurrency bugs. Because, none of the other things matter if the program is buggy. So, it is important to keep innovating new ideas and techniques to debug various types of concurrency bugs.

Concurrency bugs are software bugs related to synchronization operations of parallel programs. Parallel programs are software systems that conduct concurrent execution of multiple tasks. These tasks communicate with each other through shared memory or message passing. In this proposal, we focus on shared memory parallel programs.

Concurrency bugs occur when the programmers make mistakes in using proper synchronization operations. An example of a concurrency bug is shown in Figure 1.1. Here, two threads update a shared variable *count* without any synchronization. As a result, there can be a situation where thread T_2 reads *count*, increments it by 1 but before storing the result back to *count*, thread T_1 reads and updates *count*. As a result, the update of the thread T_1 gets lost. This is a concurrency bug namely a *Data Race*.

Data race detection has been the subject of many works (e.g., [15, 26, 45, 51, 53, 54, 55, 57, 58, 63, 65, 69, 71, 77, 78]), including the development of commercial software tools for race debugging (e.g., [26, 69]) and even the proposal of special hardware structures in the machine (e.g., [45, 57, 58, 78]). In general, there are two approaches to find data races, namely the lockset

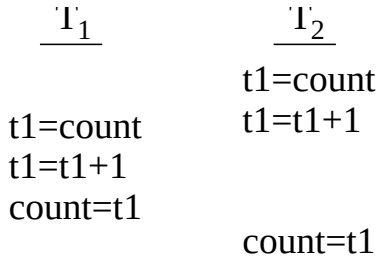


Figure 1.1: An example of a concurrency bug (data race).

approach, as in Eraser [65], and the happened-before one, as in Thread Checker [26]. The lockset approach is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Consequently, if it cannot find a common set of locks, it reports a data race.The happened-before approach relies on finding concurrent accesses by using Lamport's happened-before clock [29]. If such accesses are found, they are reported as data races. Race detectors that use these algorithms can be implemented either in software or in hardware.

A very closely related concurrency bug is an atomicity violation bug. An atomicity violation can occur when the programmer fails to enclose in the same critical section all of the memory accesses that should occur atomically. During execution, such accesses get interleaved with accesses from another thread that alter the program state, making it inconsistent.

Figure 1.2 shows an example of an atomicity violation bug in the MySql program. Variables $t \rightarrow rows$ and *binlog* need to be accessed together to generate the correct logging order of concurrent operations in the database (Figure 1.2(a)). However, the variables are protected by different critical sections. It is possible that, in between the accesses to the two variables by a thread, a second thread accesses them (Figure 1.2(b)). This is an atomicity violation, which results in a wrong logging order.

Recently, atomicity violations have got lots of attention from the researchers. Existing approaches can be classified into ones requiring explicit annotation of atomic regions (AR) and others that infer atomic regions. The proposals [22, 23, 25, 72] which are in the first category are often not applicable to the real world big software because they require lots of effort on the part of the programmers to annotate the atomic regions. Works [73, 35, 56, 39] in the second category

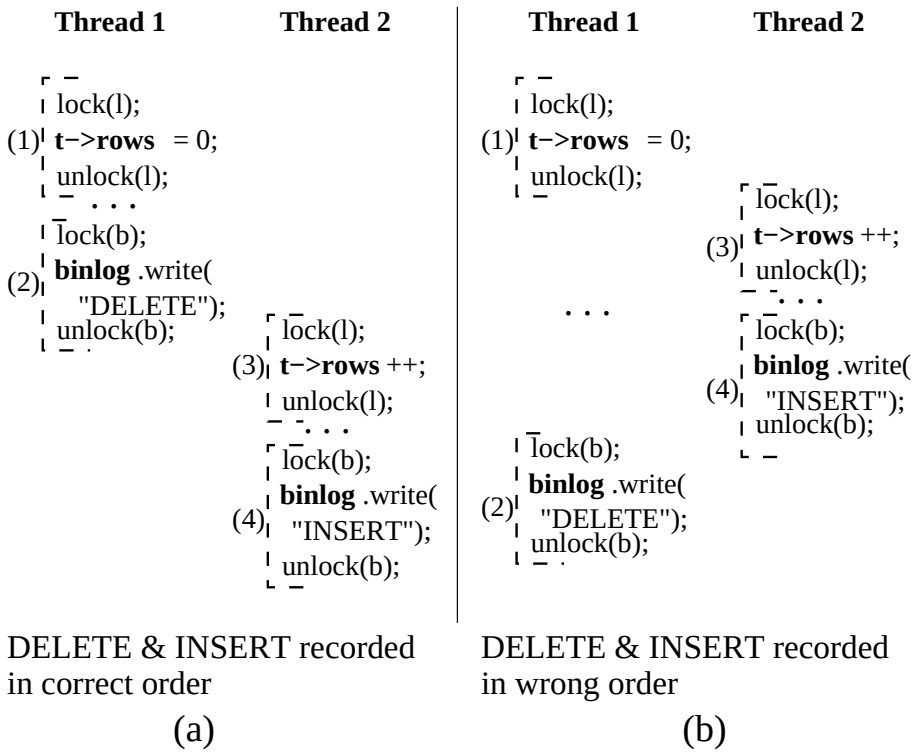


Figure 1.2: Atomicity violation in MySql. The variables involved in the bug are in bold. ,

have the nice property that it can be easily applied to any software as it does not need a priori knowledge of the atomic regions.

Among the concurrency bugs, the last but not the least one that we consider is the Sequential Consistency [28] Violation (SCV) bug. Although data races have been a topic of research for many years, not all the data races act similarly. Specifically, in terms of memory consistency, some causes violation of SC while others do not. Data races that cause SC violation are particularly difficult to detect because often times programmers use them intentionally (e.g. inside synchronization libraries) but the lack of proper memory fences causes SC to be violated. This type of data races should be given higher priority because they make it harder, if not impossible, to reason about the correctness of the program in different relaxed memory machines.

As an example, consider the simple case of Figure 1.3(a). In the example, processor P_A allocates a variable and then sets a flag. Later, P_B tests the flag and uses the variable. While this interleaving produced expected results, the interleaving in Figure 1.3(b) did not. In here, since the

variable and the flag have no data dependence, the P_A hardware reorders the statements. In this unlucky interleaving, P_B ends up using uninitialized data. This is an SC violation.

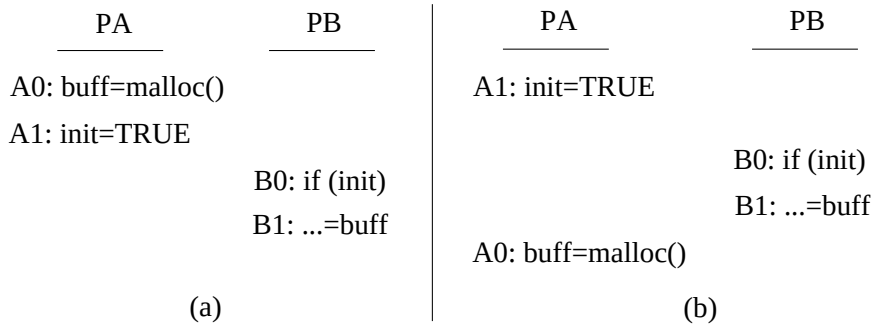


Figure 1.3: Example of an SC violation.

Most of the prior works either try to find places in the program where to insert fences to prevent SCVs [67, 18, 20, 27, 30, 70] or dynamically try to detect SC violations by looking for near by data races as a proxy for SCVs [24, 38, 43].

1.1 Proposed Approaches

We proposed three approaches to detect these three types of concurrency bugs.

SigRace [48] is proposed to detect data races with some hardware support. It relies on hardware address signatures [12]. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them to those of other processors. If the intersection is not null, a data race may have occurred. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRaces ability to detect data races.

Our second proposal, AtomTracker [49] is a comprehensive approach to AR inference and violation detection. It is the first scheme to (1) automatically infer generic ARs (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of

code construct) and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed. AtomTracker has two parts: one that automatically infers ARs (AtomTracker-I) and one that automatically detects violations of their atomicity (AtomTracker-D).

To detect SC violation bugs, we propose Vulcan. Unlike most existing approaches, it does not rely on data races to detect SC violations. Rather, it relies on the cache coherence protocol to dynamically record the observed inter-thread data dependences, and then check whether they form cycles. These dependences are kept around only for as long as they can participate in a cycle, and are discarded soon after. Both the recording and the checking of these dependences is done in hardware for minimum execution overhead.

All of these proposals detect concurrency bugs with some hardware support. The reason is that hardware support leads to low overhead solutions. Therefore, we can use them both in development time and in production run. In a nutshell, this thesis can be summarized through the following research statement.

We want to make parallel programming easier by providing always-on hardware support to detect different types of concurrency bugs.

Chapter 2

Signature Based Data Race Detection

2.1 Introduction

An important type of concurrency bug is a data race. A data race occurs when two threads access the same variable without an intervening synchronization and at least one of the accesses is a write. The erroneous program behavior caused by the race may only appear under certain access interleavings, making debugging data races notoriously hard.

For this reason, data race detection has been the subject of many works (e.g., [15, 26, 45, 51, 53, 54, 55, 57, 58, 63, 65, 69, 71, 77, 78]). In general, there are two approaches to finding data races, namely the lockset approach, as in Eraser [65], and the happened-before one, as in Thread Checker [26]. The lockset approach is based on the idea that all accesses to a given shared variable should be protected by a common set of locks. Consequently, it tracks the set of locks held while accessing each variable. It reports a violation when the currently-held set of locks (lockset) at two different accesses to the same variable have a null intersection.

The happened-before approach relies on epochs. An epoch is a thread's execution between two consecutive synchronization operations. Each processor has a logical clock, which identifies the epoch that the processor is currently executing. In addition, each variable has a timestamp, which records at which epoch the processor accessed it. When another processor accesses the variable, it compares the variable's timestamp to its own clock, to determine the relationship between the two corresponding epochs: either one logically happened before the other, or the two logically overlap. In the latter case, we have a race.

Race detectors that use these algorithms in software typically induce about 10–50x slowdowns

on programs [26, 51, 63, 65]. Such slowdowns can distort the timing of races identified in production runs, and make them hard to find. For this reason, there have been several recent proposals for race detectors with hardware assists [45, 57, 58, 78]. Such schemes should be effective at debugging races in production runs. However, they detect races by augmenting the cache state and the coherence protocol. Specifically, they tag each cache line with a timestamp [45, 57, 58] or a lockset [78], perform additional operations on local/external access to the cache, and piggyback information on cache coherence protocol messages. L1 caches and coherence protocol units are key hardware structures, either time-critical or complicated. In addition, if a line is displaced or invalidated from the cache, these systems typically lose the ability to detect races involving the line.

2.2 Contributions

We propose a novel approach to hardware-assisted data race detection that overcomes these limitations. Our approach, called *SigRace*, relies on hardware address signatures [12]. As a processor runs, the addresses of the data that it accesses are automatically encoded in signatures. At certain times, the signatures are automatically passed to a hardware module that intersects them to those of other processors. If the intersection is not null, a data race may have occurred. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races.

2.3 Background & Related Works

2.3.1 Logical Timestamps for Happened-Before

Lamport's happened-before relation [29] in a multithreaded environment states that an event α happened before another β if (i) both are performed by the same thread and α precedes β in

program order, or (ii) α is a release and β is an acquire on the same object, or (iii) for some other event γ, α happened before γ and γ happened before β. If α happened before β or vice-versa, the two events are ordered; otherwise, they are concurrent or unordered. The happened-before algorithm for race detection finds out whether two memory accesses to the same location that are performed by different threads are unordered and at least one is a write. This algorithm only detects races that actually occur during execution.

In a typical implementation, each thread maintains a logical vector clock, which has as many components as number of threads [21]. If thread *t* has a vector clock $vc_t[.]$, then the element $vc_t[t]$ contains the time of the thread itself and, given another thread *u*, $vc_t[u]$ contains the latest time of *u* "known" to *t*. When *t* performs a synchronization operation, it starts a new *Epoch* and increments $vc_t[t]$. Suppose that, after *t* performed a release on object *S*, *u* acquires *S*. In this case, *u* increments $vc_u[u]$ and, in addition, updates the rest of $vc_u[.]$ as follows: $vc_u[i] = max(vc_u[i], vc_t[i])$ for every $i \neq u$. Here, $vc_t[.]$ is the vector clock of thread *t after* the release operation. We refer to the value of a thread's vector clock during an epoch as the epoch's *Timestamp*. Figure 2.1(a) shows an example execution with epoch timestamps.

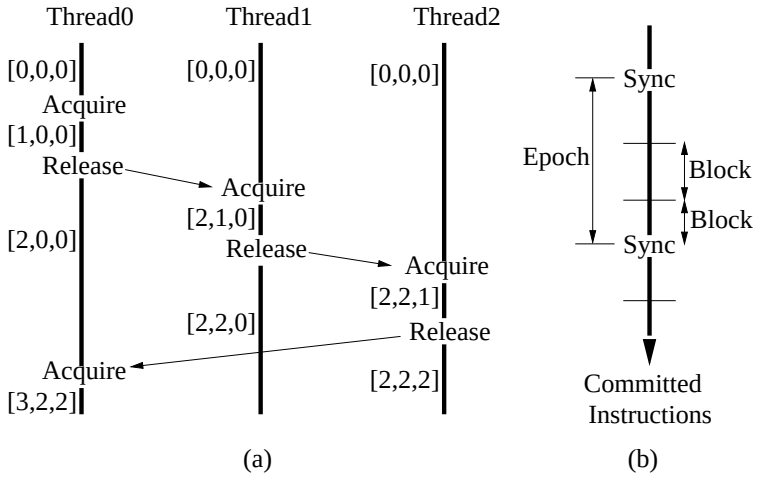


Figure 2.1: Example of execution of three threads with epoch timestamps in brackets (a), and definitions in a thread's execution (b).

We determine whether there is a happened-before relation between two epochs by comparing

their timestamps. Specifically, if epoch *f* of thread *t* has timestamp $vc_t^f[.]$ and epoch *g* of thread *u* has timestamp $vc_u^g[.]$, then *f* happened before *g* if and only if $vc_t^f[t] < vc_u^g[t]$ and $vc_t^f[u] < vc_u^g[u]$. For example, in Figure 2.1(a), the epoch after the acquire in Thread 2 happened before the epoch after the second acquire in Thread 0.

2.3.2 Hardware Schemes for Data Race Detection

There are at least four proposals for hardware-assisted data-race detectors, namely Min and Choi's [45], ReEnact [58], CORD [57] and HARD [78]. They all detect races by tagging the state in the caches as it is being accessed, and then piggybacking the tags on cache coherence protocol messages between processors so that they can be compared.

ReEnact and CORD use the happened-before approach. They tag each cache line with timestamp information, and send and compare timestamps at least at every coherence action (invalidation of cached line or external read of a dirty cached line). In ReEnact, the tag is an index into a table of vector-clock timestamps. In CORD, the tag is four scalar timestamps (two for read and two for write), and two sets of read-write bits per word. HARD uses the lockset approach and, therefore, only handles locks properly. It tags each cache line with two special state bits, and a bit vector that represents the lockset for the line. These bits are checked at every access to the line, and are kept coherent by the coherence protocol as if they were data. Finally, Min and Choi use the happened-before approach for only nested doall loops. They tag each cache line with a set of read and write bits for each doall nesting level, and perform tag checking at every cache access.

Overall, these schemes have two shortcomings. First, they modify the L1 cache, the operations performed on some local/external accesses to L1, and the cache coherence protocol messages. These are key hardware structures, either time-critical or complicated to design and debug. Second, when a line is displaced or invalidated from the cache, the system loses its ability to detect a data race for that line. An exception is CORD, which keeps some timestamp information in memory. We would like a design that decouples cache and coherence protocol from race detection, and has a longer detection window than that provided by cache residence.

2.3.3 Hardware Address Signatures

A hardware address signature is a long register (e.g., 2Kbits long) where the memory addresses accessed by the processor are automatically hash-encoded and accumulated using a Bloom filter [5]. Signatures have been used in the Bulk system [12] and several subsequent multiprocessor designs (e.g., [11, 46, 74]) to detect data dependences between threads in thread-level speculation and transactional memory. Signatures are efficiently operated on in hardware using simple logic (e.g., bit-wise AND of signatures to find common addresses). From a signature, it is only possible to obtain a superset of the addresses that were originally encoded in the signature. Consequently, operations on signatures may produce false positives, although not false negatives.

In this work, we use signatures to detect data races. This is the first proposal that uses address signatures for happened-before race detection.

2.4 Signature Based Race Detection

2.4.1 Overview of the Idea

The idea in SigRace is to automatically record the set of addresses accessed by the processor in a code section in hardware signatures. At appropriate intervals, the signatures and the epoch timestamp are automatically passed to an on-chip hardware module called *Race Detection Module* (RDM). The RDM keeps the signatures and the timestamp in an in-order queue assigned to the initiating processor, and compares them to the entries of queues assigned to other processors using very efficient signature operations. The comparison quickly determines whether there has been a potential data race.

SigRace addresses the two shortcomings of existing hardware-assisted schemes. First, there are no L1 cache modifications, no critical-path operations performed on local/external accesses to L1, and no cache coherence protocol message changes. Signature generation, storage, and comparison are decoupled from caches and coherence protocol. Second, lines can be displaced

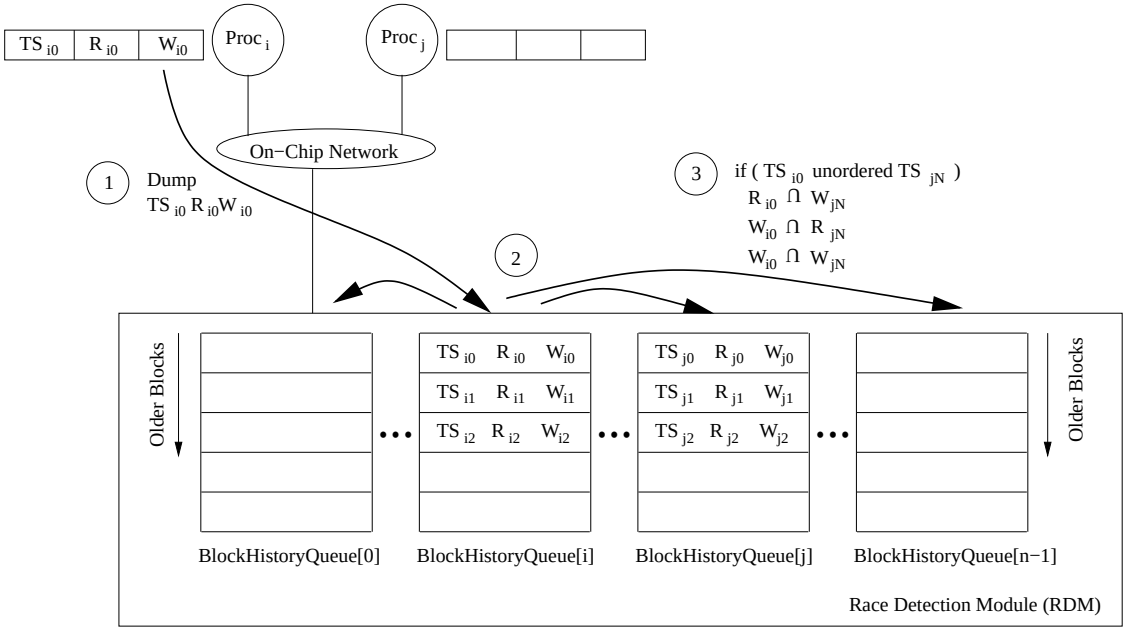


Figure 2.2: Operations when a block finishes. In the figure, *TS*, *R*, and *W* refer to timestamp and read and write signature, respectively. In any BlockHistoryQueue[k], entries for older blocks have higher subscripts.

or invalidated from caches without SigRace losing the ability to detect data races. In practice, the RDM necessarily has limited storage capacity, and old signatures are discarded when room is needed, also limiting the race detection window. We see, however, that SigRace's race detection capability is higher than that of cache-based systems.

SigRace needs to rely on rollback and re-execution to provide the full set of racing instructions to the programmer, and to disambiguate false-positive races. Unlike currently-proposed schemes, SigRace does not suffer false positives due to false sharing. This is because SigRace encodes *fine-grain* (e.g., word) addresses in signatures. Accesses to different words of the same line do not induce a data race report. However, address aliasing in signatures may induce false positives in SigRace. This is because signatures represent a superset of the addresses that were encoded [12]. False negatives are not possible.

For simplicity, we want SigRace to support the rollback and re-execution largely in software. Consequently, SigRace does not use thread-level speculative execution. Reads and writes commit as usual. We use the ReVive checkpointing/rollback mechanism proposed by Prvulovic *et al.* [59].

After rollback and re-execution to the race, an analysis phase takes place. We envision rollback, re-execution, and analysis to be transparent to the user, who should at worst notice a slight slowdown when many false data races are detected.

Address collection into signatures is disabled and enabled in software at kernel entries and exits, respectively, and, optionally, at library entries and exits. This typically improves race detection. Moreover, the programmer can disable address collection during the execution of certain code sections. Finally, signatures are assigned to software threads rather than to hardware contexts.

In the following, we describe SigRace's operation under three stages: normal execution, re-execution, and race analysis.

2.4.2 Normal Execution under SigRace

The execution of a thread is logically divided into epochs, which are the dynamic instructions executed between synchronization operations (Figure 2.1(b)). The latter include, e.g., acquiring a lock, releasing it, waiting on a flag, setting a flag, or crossing a barrier. Under SigRace, each processor keeps the timestamp of the current epoch, which is encoded and updated as per Section 2.3.1. In addition, the processor has a Read (R) and a Write (W) Signature. When a load or a store commits, a hardware Bloom filter as in [12] automatically hash-encodes and accumulates the address loaded from or stored to, respectively, into the correct signature.

Ideally, a processor can keep its timestamp and R and W signatures to itself until the end of the epoch. At that point, they are made visible to all other processors, to check for data races. In practice, long epochs would cause the signatures to accumulate so much state that any operation on them would likely induce many false positives due to aliasing [12]. Consequently, when the processor has committed a certain number of dynamic instructions that we call a *Block* without finding a synchronization operation, the hardware automatically passes the timestamp and signatures to the RDM. Figure 2.1(b) shows the resulting execution: a block finishes when either a certain number of dynamic instructions have been committed or a synchronization operation is found.

The exact actions taken when a block in processor *i* finishes for either reason are as follows (Figure 2.2). First, the hardware automatically dumps the timestamp and R and W signatures into a memory-mapped FIFO queue of registers in the RDM called BlockHistoryQueue[i] (Step *1* in the figure). To save network bandwidth, the data is transferred in compressed format. The R and W signatures are then cleared. Finally, if the block finished because of a synchronization operation, library software updates the epoch timestamp and then saves it in a log in memory to keep a trail of timestamp changes — which is useful if we need to roll back execution.

At the RDM, simple hardware automatically compares the incoming data to entries in all the other BlockHistoryQueue[.] (Step *2* in the figure). Specifically, for a given BlockHistoryQueue[j], the incoming timestamp TS_{i0} gets compared to TS_{j0}, TS_{j1}, etc — in sequence order starting from the latest one available. Such comparisons stop as soon as one of the *j* timestamps is found to precede the incoming timestamp — in this case, due to transitivity, all earlier *j* timestamps will also precede the incoming one. Then, for all timestamp pairs found to be unordered (e.g., TS_{i0} and TS_{jN}), simple signature functional units compute $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ (Step *3* in the figure). If any of these is not null, the two blocks have accessed the same location(s) without synchronization and at least one wrote. We have detected a data race — or a false positive. We call these two blocks and their corresponding threads the *Conflicting Blocks* and *Threads*.

A BlockHistoryQueue[k] is a FIFO queue. When it overflows, information on the displaced blocks is lost. We have lost the ability to detect data races in those blocks. We accept this limitation to keep overheads to a minimum.

2.4.3 Re-Execution under SigRace

When a pair of Conflicting Blocks is found, we want to identify for the user the exact instructions and address(es) involved in the race(s), and to weed out any false positive transparently to the user. In our design, an exception forces all processors to roll back to the previous checkpoint and enter the *Re-execution* mode. In this section, we describe the checkpointing support and the re-execution process.

Checkpointing Support

The SigRace design that we present needs a low-overhead checkpointing scheme. Ideally, such a scheme would already be in place for reliability purposes, and SigRace would reuse it. One possible scheme is ReVive [59], which performs incremental memory-state checkpointing.

In addition, the kernel collects and buffers the inputs to the program during Normal execution — such as interrupts, system call returns, and I/O input — and passes them to the re-execution at appropriate times. Support similar to this is provided by Flashback [68] and Rx [60], which require no hardware modifications.

With these two mechanisms, we will now see that SigRace re-executes following the same paths until the first data race is found.

Re-Execution Operation

Re-execution forces the application to follow the same order of epochs as in the original execution, and leaves each thread at the beginning of the *epoch* that the thread was executing when the race was detected. This is shown in Figure 2.3, where a race was detected at the points shown in Figure 2.3(a), and re-execution brings the threads to points *A*, *B*, *C*, and *D* in Figure 2.3(b). Note that re-execution does not bring each thread to the actual block that it was executing when the race was detected. This is because we do not rely on the ability to reproduce block boundaries exactly.

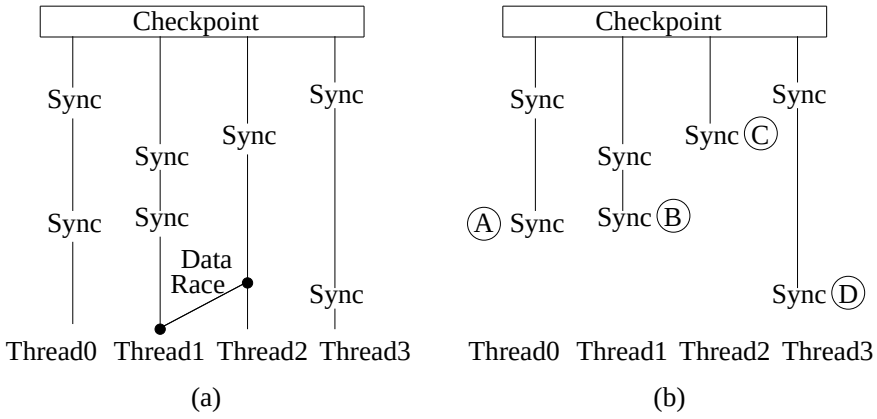


Figure 2.3: Detection of a data race during Normal (a) and Re-execution (b) modes.

To reproduce the order of epochs, SigRace uses the history of logged timestamps (Section 2.4.2). They encode the history of synchronization operation orders — i.e., which thread completed a synchronization operation before which other thread. SigRace uses these timestamps to follow the same synchronization orders.

Specifically, each processor has a Thread Re-execution Timestamp (TRT) register into which, as it re-executes, it successively loads the timestamps logged since the checkpoint. Recall that each timestamp was saved *after* the processor went past a synchronization operation. In addition, there is a shared software structure in memory called Global Re-execution Timestamp (GRT) that contains the most up-to-date logical time of each processor during the re-execution. In other words, while the TRT is the "thread view" of the current re-execution time, the GRT is the "true global view". Each processor compares its TRT to the GRT to see when the other processors have executed all the earlier epochs and the processor can proceed. Proceeding means for the processor to perform its next synchronization operation, update its own component of the GRT, execute its next epoch, and read its next logged timestamp into its TRT.

The actual algorithm is as follows. Let us call $grt[.]$ the GRT and $trt_p[.]$ the TRT of processor p. Each i in $grt[i]$ is the latest epoch from processor i that has been executed. For example, Figure 2.4 repeats the timeline of Figure 2.1(a) and shows with an arrow the current position of each replaying processor. As a result, $grt[.] = [2, 1, 0]$. All processors are waiting at a synchronization operation and we need to decide which one(s) to execute next. Each processor has loaded into its trt the timestamp it had *after* the synchronization (e.g., $trt_1[.] = [2, 2, 0]$). When a given processor p finds that $grt[i] \geq trt_p[i]$ for all $i \neq p$, then processor p executes the synchronization operation, sets $grt[p] = trt_p[p]$, executes its next epoch, and loads its next logged timestamp into $trt_p[.]$. The last two operations are not performed if there is no next logged timestamp. In the figure, the only processor for which the inequality is true is Processor *1*. Consequently, Processor *1* will execute the release and set GRT to [2,2,0]. Since it has no further timestamp logged, it will wait there.

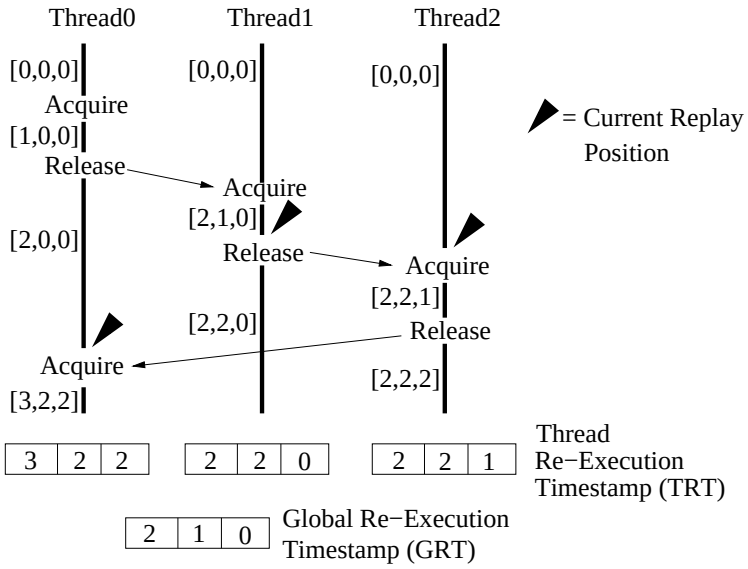


Figure 2.4: Re-execution using the logged timestamps.

2.4.4 Race Analysis under SigRace

When all threads have reached their last logged timestamp, execution enters the *Analysis* mode. In this mode, only the threads involved in the data race execute, while the others stall. Specifically, first, the two processors executing the Conflicting threads load into a local register called the *Conflict Signature* the intersection of the two Conflicting blocks' signatures — namely the union of $R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, and $W_{i0} \cap W_{jN}$ as per Section 2.4.2. The Conflict Signature holds the hashed address(es) involved in the race. Then, the two Conflicting threads execute normally up to their next synchronization points, while the hardware automatically intersects their loads and stores against the Conflict Signature. Every time a non-null intersection occurs, a trap is triggered, which records the memory address and the PC. Finally, when both threads have reached their next synchronization points, a software handler compares the record of trapping addresses in both processors, to see if there are common addresses. If so, SigRace has found a data race, which it reports to the user. Otherwise, it was only a false positive and is ignored.

As each Conflicting thread reaches its next synchronization point, it may have executed past its Conflicting block. This is fine, since it enables us to capture as many of the references involved in the data race(s) as possible.

After the Analysis step, execution *seamlessly* returns to the Normal mode of execution. This is enabled by the fact that SigRace continued to perform timestamp/signature logging and signature intersection during Re-execution and Analysis modes — exactly like it did during Normal mode. In this way, the trail of timestamps and signatures is up to date at the point where Analysis completes and all processors resume Normal execution.

Because the Analysis step may push program execution beyond what was executed before the rollback, it is possible that the Analysis step discovers new data races. To address this case, SigRace proceeds as follows. Every time two blocks are found to conflict during Analysis ($R_{i0} \cap W_{jN}$, $W_{i0} \cap R_{jN}$, or $W_{i0} \cap W_{jN}$ are not null), a handler compares their intersection against the contents of the Conflict Signature. If the latter is a superset, no action is taken because this race is already being processed (call it *Race1*). Otherwise, the handler saves the signature intersection and records the need to analyze the new data race (call it *Race2*) later. In this case, after *Race1* is fully analyzed, execution is rolled back, and we proceed to perform Re-execution and Analysis for *Race2*. Note that we cannot analyze the two races concurrently because, by the time we detect the presence of *Race2*, processors have already issued some of the references associated with it.

Overall, to minimize the amount of re-execution, SigRace is designed as follows. When a processor in Normal execution detects a pair of Conflicting blocks, it does not immediately request a rollback. Instead, it continues executing for several more blocks (e.g., 5–10) or until it synchronizes, before interrupting all other processors and requesting rollback. The goal is to collect as many potential races as possible. During Analysis, the Conflict Signature of each processor contains the racing addresses of all the races that the processor is involved in characterizing. In this way, multiple races are analyzed concurrently. Finally, SigRace also saves the Conflict Signatures of the races that it has finished analyzing. In this way, if SigRace has to re-execute the same code a second time, it can ignore the race already analyzed.

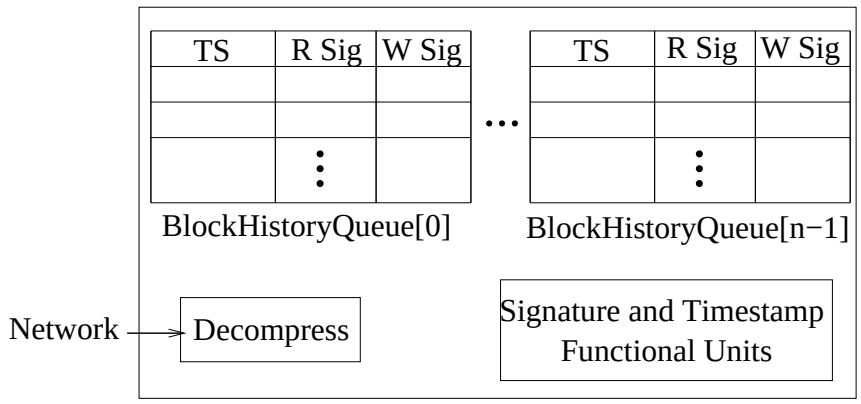
2.5 Implementation

A possible implementation of SigRace requires some hardware and software changes to a chip multiprocessor. The hardware changes are the Race Detection Module (RDM) and some additions to the per-processor cache hierarchy. The cache tag and data arrays are *unmodified*. Also, SigRace does not use speculative multithreading. On the software side, SigRace needs an augmented synchronization library. In this section, we describe the hardware and software components, and then how SigRace is virtualized to make it usable.

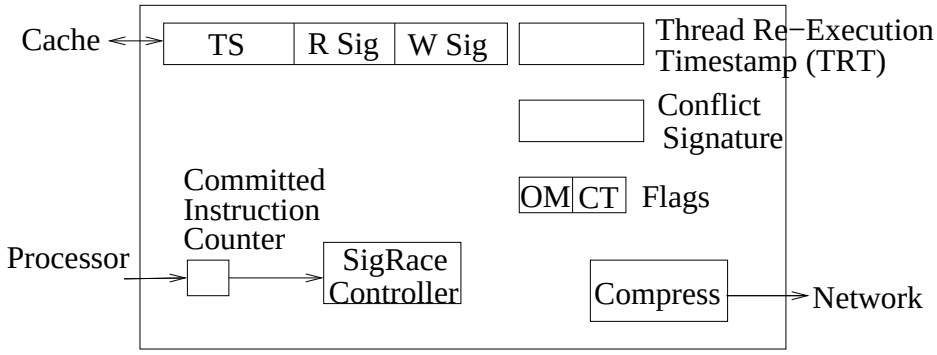
2.5.1 Hardware Modifications

The RDM is a simple on-chip hardware module that is connected to the on-chip network. As shown in Figure 2.5(a), it contains the BlockHistoryQueue[.], which stores past timestamps (TS) and signatures for all the processors (Section 2.4.2). It also includes functional units that operate on signatures (like in Bulk [12]) and timestamps.

SigRace also requires some per-processor hardware that is placed in the cache hierarchy in a module that interfaces with the processor, the cache and the network (Figure 2.5(b)). The module includes storage for the current epoch timestamp and the current block's R and W signatures. The addresses hashed into signatures have a finer granularity than cache line, so that false sharing of a line does not trigger incorrect data race alarms. A good choice is to use word addresses. The module also includes the Thread Re-Execution Timestamp (TRT) for re-execution (Section 2.4.3) and the Conflict Signature for analysis (Section 2.4.4). There are two flags, namely the *Operation Mode* (OM) that denotes whether the hardware is in Normal, Re-execution, or Analysis mode, and the *Conflicting Thread* (CT) that denotes whether the thread is a Conflicting one (Section 2.4.2). There is also a *Committed Instruction Counter*. When the latter reaches the maximum value set for a block — or an approximate value, since there is no need to be exact — it sends a signal to terminate the current block. The SigRace controller then initiates the following actions: dump TS, R and W into the corresponding BlockHistoryQueue[i], and clear R, W, and the Committed



(a) Race Detection Module (RDM)



(b) Additions to the Private Cache Hierarchy

Figure 2.5: Hardware support needed by SigRace.

Instruction Counter.

The TS and R and W signatures are compressed before being sent to the on-chip network, and decompressed as they get into the RDM. We call these network messages the *Summary* messages. Their compressed size is ≈100 bytes — for 2 signatures of 2 Kbits each and a 160-bit timestamp. This is less than the size of two cache lines, and is sent out every time a block completes (≈2,000 committed instructions). Summary messages from the same processor need to arrive at the RDM in order; messages from different processors can arrive in any order. This centralized RDM design is fine for the small numbers of processors considered here. In large, distributed machines, the RDM can be distributed as well.

2.5.2 Software Interface

High-level synchronization constructs such as M4 macros [41] and OpenMP directives [16] are commonly used by programmers and parallelizing compilers. These constructs can enable SigRace transparently. Specifically, we rewrite such constructs to encapsulate the SigRace operations. As a result, the application code does not need be modified at all, and all we need is to relink it with the new M4 or OpenMP library.

To accomplish this, we add three processor instructions that operate on local SigRace structures (Table 3.1). Two of the instructions (*collect_on* and *collect_off*) enable and disable the collection of addresses into signatures, and the counting of committed instructions. A variation of these instructions could perform these actions only on a range of addresses. These instructions are used to prevent the signatures from being polluted by unrelated accesses (such as those from the OS or the instrumentation added to the macros) or by obviously-private accesses (e.g., those to the stack). They can also be used to mark a benign data race or an epoch that should *skip the checking*. The other instruction (*sync_reached*) is invoked when execution reaches a synchronization operation. Specifically, it is invoked immediately before performing a release-type operation and immediately after performing a successful acquire-type operation. It tells the SigRace controller to dump TS, R and W into the RDM, clear R, W, and the Committed Instruction Counter, and increment the counter in TS that corresponds to the local thread.

Instruction	Description
collect_on	Collect addresses into R and W, and count committed instructions.
collect_off	Do not collect addresses into R or W, or count committed instructions.
sync_reached	Dump TS, R, and W into the RDM. Clear R, W and the Committed Instruction Counter. Increment the counter in TS that corresponds to the local thread.

Table 2.1: Instructions to manage SigRace structures.

For simplicity, we assume that these instructions make their side effects visible only when they

commit — like the updates of signatures by loads and stores. A design where these actions happen earlier in the pipeline can also be conceived.

2.6 Conclusions

We propose SigRace, a novel approach to hardware-assisted data race detection that overcomes shortcomings of previous hardware proposals. To detect races, SigRace does not rely on cache state or coherence protocol messages. Instead, it relies on hardware address signatures. With SigRace, there are no changes to the cache or the cache coherence protocol messages, and there are no critical-path operations performed on local/external access to the cache. Moreover, lines can be displaced or invalidated from caches without affecting SigRace's ability to detect data races. Finally, application code is unmodified. Our experiments showed that SigRace is significantly more effective than a state-of-the-art conventional hardware-assisted race detector. SigRace found on average 29% more static races and 107% more dynamic races. Moreover, if we inject data races, SigRace found 150% more static races than the conventional scheme.

Chapter 3

A Comprehensive Approach to Atomic Region Inference and Violation Detection

3.1 Introduction

Of all the concurrency bugs, atomicity violations are particularly hard to isolate, and have received little attention compared to their importance [33]. An atomicity violation can occur when the programmer fails to enclose in the same critical section all of the memory accesses that should occur atomically. During execution, such accesses get interleaved with accesses from another thread that alter the program state, making it inconsistent.

Existing approaches to find these bugs can be classified into two groups. The first one are techniques that require the programmer to annotate the Atomic Regions (AR) [22, 23, 25, 72]. Providing this information may be too tedious and error prone on the part of the programmer. The second group are techniques that attempt to detect atomicity violations automatically. They include, among others, SVD [73], AVIO [35], AtomFuzzer [56], and Atom-Aid [39].

These techniques are often effective. However, as we will discuss in detail, they are all constrained in the types of ARs that they can support — typically limited by the number of variables that they access, the number of instructions that they execute, or the type of code construct in which they are embedded (e.g., a function). For example, AVIO only finds ARs with two instructions and a single variable. A substantial improvement in the state of the art would be to devise an approach that identifies violations of *any* type of AR.

3.2 Contributions

We propose *AtomTracker*. which is a comprehensive approach to AR inference and violation detection. It is the *first* scheme to (1) automatically infer *generic* ARs (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of code construct) and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed.

AtomTracker has two parts: one that automatically infers ARs (*AtomTracker-I*) and one that automatically detects violations of their atomicity (*AtomTracker-D*). AtomTracker-I infers generic ARs by analyzing annotation-free memory traces of test runs of the program. AtomTracker-I's main contribution is a novel algorithm that works by greedily joining successive references of a thread into an AR if the other threads do not conflict. AtomTracker-I does not require any semantic knowledge of the program. It is the first algorithm of its kind.

AtomTracker-D takes the set of ARs and detects violations of their atomicity at runtime. AtomTracker-D's first contribution is an algorithm for atomicity violation detection. It checks if concurrently-executing ARs can be made to appear to execute in sequence by taking one reference at a time and reconsidering the relative order of the ARs. The second contribution is a *hardware implementation* of AtomTracker-D in a shared-memory multiprocessor that leverages cache coherence state transitions. It induces a negligible execution time overhead and, therefore, can be on during production runs.

3.3 Related Works

The state of the art in atomicity violation detection without annotations is set by SVD [73], AVIO [35], MUVI [32], AtomFuzzer [56], PSet [75], and Bugaboo [36]. SVD [73] proposes the Computational Unit (CU) concept, which approximates a limited type of AR. The idea is that, after a thread has written to a shared variable, when it reads it again, it starts a new CU. As a program runs, SVD computes CUs based on the observed dependencies. SVD reports violations

when CUs are interleaved with unserializable writes from other threads. While SVD is effective, it only looks for violations of the limited set of ARs considered.

AVIO [35] looks for ARs composed of only a single variable and two instructions. AVIO uses memory traces of correct executions of the program to train the algorithm. If two instructions in a thread that access the same shared variable are never found to be interleaved unserializably by an access from another thread while training, then AVIO assumes that these two instructions are intended to be atomic by the programmer. Consequently, AVIO reports violations when these instructions are interleaved unserializably in production runs.

MUVI [32] is a step toward handling multiple variables. It finds access correlations among multiple variables. Variables that are accessed together for some minimum number of times are likely to be related. These variables should be protected by the same lock. The MUVI paper shows how to use correlation information in a race detector. It claims that this information would be hard to use to detect multi-variable atomicity violations.

AtomFuzzer [56] looks for the case when a lock is grabbed multiple times in the same function. It reckons that this pattern suggests an atomicity violation bug. This is because functions are likely to be atomic.

PSet [75] detects and avoids concurrency bugs by embedding in the binary legal interleavings obtained from training runs. This is done by specifying which memory operation depends upon which other remote memory operations. If, at runtime, an unexpected interleaving is observed, the system reports a potential bug. Bugaboo [36] extends this work by adding context information to the interleavings. A given interleaving is acceptable under an certain program context, while it is not under a different one. These two works do not specifically target atomicity violations, but can be used to detect unusual interleavings that uncover such violations.

A recent work that can detect multi-variable atomicity violations is ColorSafe [37]. This scheme requires *annotations* of which groups of variables are related (i.e., have the same color). Then, the system reports a violation when a thread's accesses to same-color variables are interleaved by another thread's access to a variable of that color.

While the work in this area has made great strides, it is still limited, considering the many dimensions of the problem. Our scheme, AtomTracker, differs from these proposals in that it is the first one to work with annotation-free *arbitrary* (non-nested) ARs. Such ARs are not constrained in the number of variables, number of instructions, or type of code construct (e.g., a function). Moreover, AtomTracker both automatically *infers* ARs in a program (AtomTracker-I) and automatically *detects* atomicity violations of these ARs at runtime (AtomTracker-D). All this makes AtomTracker the first of its kind.

3.4 AtomTracker-I: Automatic Inference of Atomic Regions

AtomTracker-I automatically infers ARs from a program by training on the memory traces of many correct executions of the program. Training on correct runs of programs is a feasible and commonly-used approach in software-development groups. It is also used in many proposed debugging tools, such as AVIO [35] or PSet [75].

AtomTracker-I does not require any manual annotation from the programmer. The key idea is to scan the dynamic memory reference trace of a thread, and *greedily try to join* successive references of the thread into a common AR — for as long as the references of the other threads do not conflict with this newly-formed AR. The operation is repeated for the references of each thread in turn, and then for each training file. The output of AtomTracker-I is a list of static AR entry and exit points in the program.

With this algorithm, the analysis of the first trace file typically generates a set of large ARs. Later, as we process another trace file, we may find evidence that an AR should be broken into two or more smaller ARs. As we process more files, ARs will tend to get more numerous and smaller. When the set of ARs does not change anymore, we assume that we have found the actual ARs. The rationale is that if a set of accesses by a thread are found to be atomic in all the correct runs, then the programmer likely intends them to be atomic. As we can see, this algorithm extracts any arbitrary AR, unconstrained in the number of variables, number of instructions, or type of code

construct.

Next, we describe the basic algorithm, some design decisions, and two examples.

3.4.1 Basic AtomTracker-I Algorithm

AtomTracker-I processes multithreaded traces of a program's memory accesses. A trace file has an ordered list of records from several threads collected during an execution of the program. Each record contains the thread ID, PC, address accessed, and whether it is a load or a store. AtomTracker-I processes many trace files of correct runs.

The algorithm works by greedily trying to join successive references of a thread into an AR. The goal is to generate ARs that are as big as possible. To describe the algorithm, we use the two-threaded trace of Figure 3.1(a). In the figure, I_i:*rd x* means that the instruction at PC I_i reads address *x*.

The algorithm starts by processing all the references of one thread, then moving to a second thread, and so on. The order of thread processing may initially generate small variations, but the end result is the same.

In the example, we start with Thread 1. The thread reads *x* in I_1 and then *y* in I_2. We would like to group I_1 and I_2 in an AR. However, Thread 2 writes *y* in between (in J_1). We cannot assume that I_2 happened before J_1 (and, therefore, move I_2 above J_1) because, if we do, Thread 1 would read a wrong *y* value. However, we can assume that I_1 happened after J_1 without affecting the final execution outcome. Consequently, we move I_1 down after J_1 (hence, the arrow) and combine I_1 and I_2 into an AR called $I_{1'}$ that reads *x* and *y*. Conceptually we think of it as a giant instruction. Next, we consider combining $I_{1'}$ with I_3. We cannot move $I_{1'}$ below J_2 of Thread 2 because there is a conflict on variable *x*. We cannot move I_3 upward above J_2, either. Therefore, AR $I_{1'}$ cannot grow any bigger (Figure 3.1(b)).

We start a new AR with I_3. Applying the same algorithm, we move I_3 down and combine it with I_4. Then, we move the $I_3 + I_4$ combination down, combine it with I_5 and get the bigger AR $I_3 + I_4 + I_5$. We call this AR $I_{5'}$ (Figure 3.1(c)). Now, we cannot move $I_{5'}$ down or move I_6 above

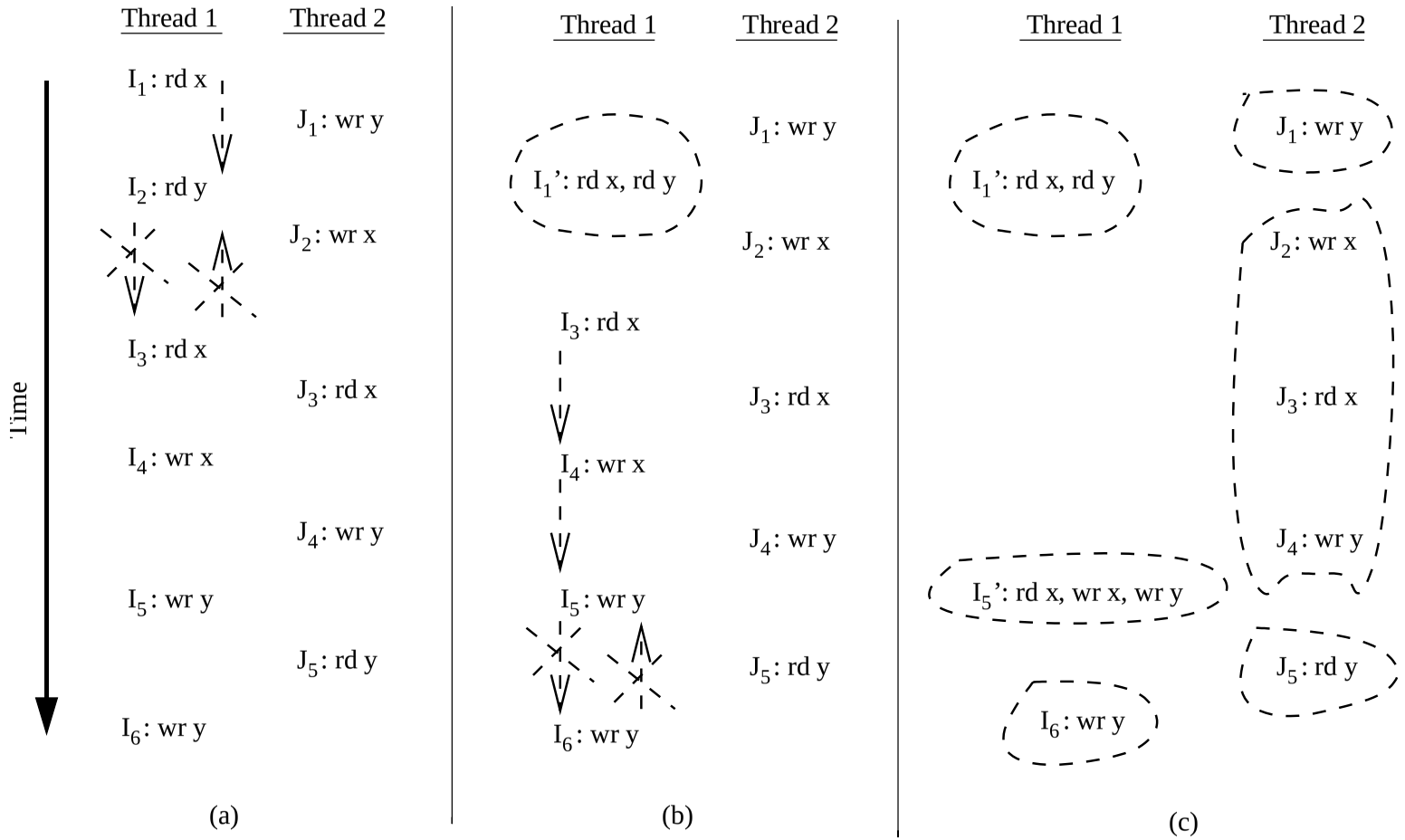


Figure 3.1: Basic AtomTracker-I algorithm for inferring ARs.

J_5 because of the conflict with y. So, $I_{5'}$ does not grow any more.

If the trace contains more than two threads, every time we attempt to combine two accesses in Thread 1, we need to consider the intervening accesses from *all* the other threads. After processing Thread 1, we move to process Thread 2, then Thread 3, and so on. In the example, since there are only 2 threads, the ARs in Thread 2 are created as a side effect of processing Thread 1 (Figure 3.1(c)).

As AtomTracker-I processes a trace file, it may find that an AR that was previously inferred from the same file gets divided into multiple ARs. Therefore, after AtomTracker-I finishes the file, it goes back to re-process it from the beginning, and breaks the previous AR into the multiple smaller ones. This process continues iteratively until AtomTracker-I gets a stable set of ARs from this trace file.

After this, AtomTracker-I takes these ARs and moves to analyze the traces of another run. As AtomTracker-I processes the new file, it starts with the ARs obtained from previous files and likely breaks some of them into smaller ones. The process is repeated until the set of total ARs does not change by analyzing more runs. The fact that every iteration can only create smaller ARs ensures that AtomTracker-I always converges.

AtomTracker-I could also work even if some of the training runs were of incorrect executions. In this case, after the ARs are collected, we would apply statistical analysis to identify the good ARs. Specifically, if a certain AR appeared at least a threshold number of times in the test runs, we would consider it a good AR; otherwise, we would discard it.

3.4.2 Design Decisions

Handling Synchronization Variable Accesses

When AtomTracker-I attempts to move references up and down, it disregards synchronization accesses. This prevents synchronization bugs in the program from making AtomTracker-I infer incorrect ARs.

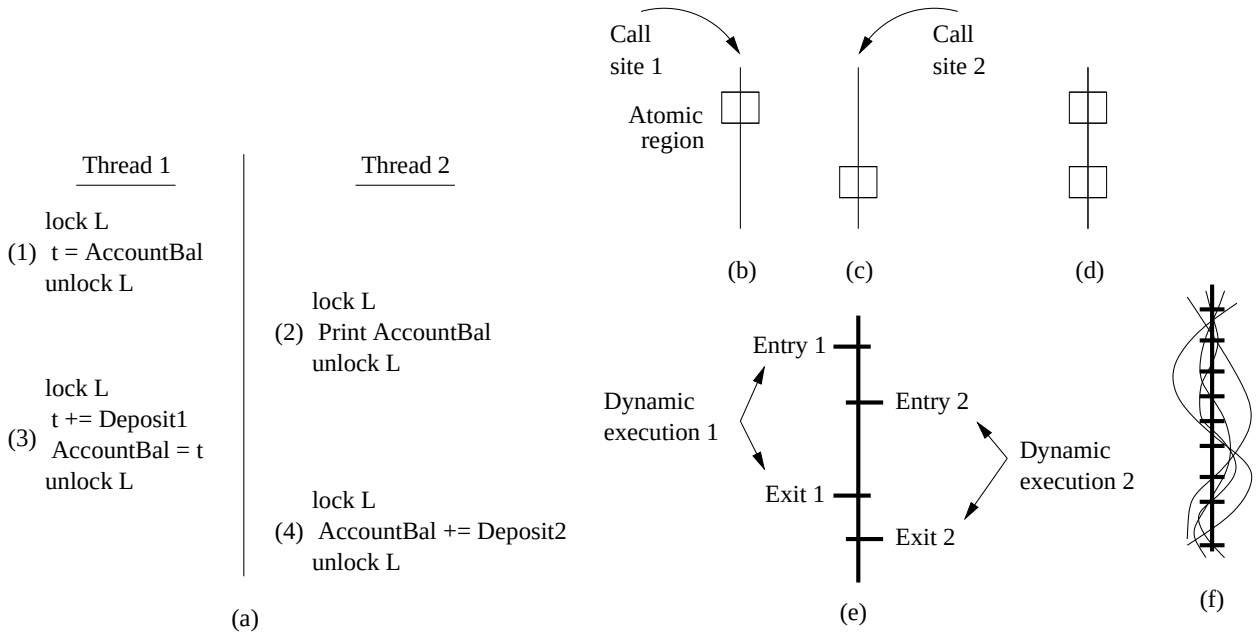


Figure 3.2: Design decisions in AtomTracker-I.

To see why, consider Figure 3.2(a). Thread 1 incorrectly uses two separate critical sections to add *Deposit1* to *AccountBal* — one to read the balance *(1)* and one to write it *(3)*. During training runs, a harmless critical section that prints the balance *(2)* interposes between the two. If we made the synchronization accesses visible to AtomTracker-I, it would detect conflicts on lock variable *L*, and incorrectly create separate ARs for *(1)* and *(3)*. Later, during production runs, if critical section *(4)* interposed between *(1)* and *(3)*, we would not detect an atomicity violation. However, by disregarding synchronization accesses, AtomTracker-I moves code *(1)* below *(2)*, and correctly merges *(1)* and *(3)* into the same AR.

Using Critical Section Information

Programmers mark as critical sections portions of the code that they want to be atomic. Typically, an atomicity violation occurs because the critical section that the programmer wrote is not large enough, although what is inside should indeed be atomic. Consequently, AtomTracker-I uses the lock/unlock statements in a program as a hint that they enclose code that should not be split into multiple ARs.

Specifically, the AtomTracker-I algorithm includes a preprocessing pass on the trace files. The pass identifies each lock/unlock pair that protects a critical section whose data is not being accessed concurrently by another thread. The sections protected by such lock/unlock pairs are recorded in a table. Later, when the AtomTracker-I algorithm runs, it uses this table. Specifically, it considers each of the sections in the table as an indivisible instruction. When it tries to expand an AR and finds one of these critical sections, it either takes-in the whole section or no instruction from it.

Using Loop Information

Typically, a programmer either makes a whole loop atomic or creates one or multiple ARs within an iteration of the loop. ARs very rarely cross an iteration boundary in a loop. To exploit this fact, in the same preprocessing pass described above, AtomTracker-I also identifies loops and checks whether they conflict with concurrent accesses from other threads. If none of the instances of a loop has conflicting accesses, then AtomTracker-I stores the loop boundaries in a database for later use. Later, when the AtomTracker-I algorithm runs, it will consider the whole loop as one giant indivisible instruction; when it tries to expand an AR, it either takes in the whole loop or no instruction from it.

On the other hand, if the loop has conflicting accesses in the trace, then the AtomTracker-I preprocessing pass records the iteration boundaries of the loop in the database for later use. Later, when the AtomTracker-I algorithm runs, it will always end an AR at the iteration boundary, so that no AR crosses the boundary. This, of course, does not disallow multiple ARs within an iteration.

To keep things simple, we flatten the inner loops and only consider outer loops. To make the preprocessing pass possible, loops are dynamically identified automatically using the algorithm of Moseley *et al* [47] during training runs, and this information is recorded in the trace files.

Handling Different Call Sites

Depending on what site a subroutine is called from, it may behave differently and, therefore, AtomTracker-I may create different ARs. For example, assume that when a subroutine is called

from Site 1, AtomTracker-I creates the AR in Figure 3.2(b), while when it is called from Site 2, it creates the one in Figure 3.2(c). For simplicity, we make AtomTracker-I context insensitive. This means that, for each subroutine, we use the combination of all the ARs found in all of the calls. In the example, we use the ARs in Figure 3.2(d).

Atomic Region Representation

The goal of AtomTracker-I is to augment the *static* code of the program with AR entries and exits. This is not trivial because each training run may take different control paths.

For example, Figure 3.2(e) shows a segment of static code and two dynamic executions that took different control paths and found different AR entries and exits. Pictorially, Figure 3.2(f) shows many dynamic executions (shown as curved lines) that intercept the static code differently, inserting different AR entries and exits.

To ensure that the resulting information makes sense to AtomTracker-D, the AR exit markers inserted by AtomTracker-I in the code also include the PC of the corresponding AR entry point. In this way, if dynamic execution *2* in Figure 3.2(e) finds Exit *1*, it ignores it because it is not paired with Entry *2*.

3.4.3 Putting It all Together

To summarize, this is how the complete AtomTracker-I algorithm works. When first presented with a trace file, AtomTracker-I performs a preprocessing pass to record lock/unlock pairs as per Section 3.4.2 and loop boundaries and iteration boundaries as per Section 3.4.2. Then, AtomTracker-I runs the algorithm of Section 3.4.1, as many times as it needs to converge (typically 2–4 times).

The reason why the algorithm may need to run multiple times over the same trace is because key information for AR inference may only appear toward the end of the trace. This is seen in the trace of Figure 3.3. Suppose that the correct ARs for Thread 1 are those in Figure 3.3(a). However, as AtomTracker-I eagerly builds the first AR, such AR will gobble-up the I_3 read to x — since Thread 2 only reads x. AtomTracker-I will not include the I_4 write to y in the same AR

because J_2 from Thread 2 conflicts. Consequently, AtomTracker-I will record that I_3 is the end of the AR starting at I_1 (Figure 3.3(b)).

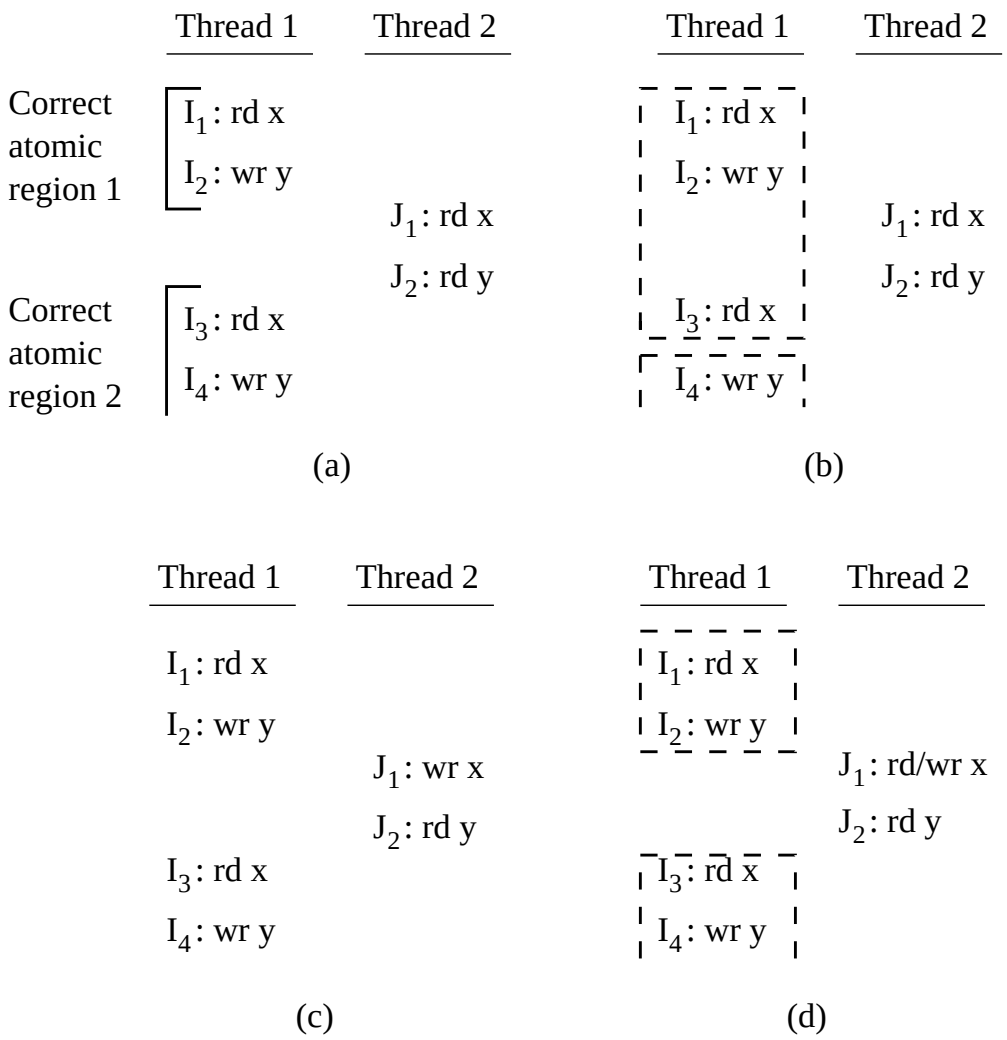


Figure 3.3: ARs correct themselves thanks to the multiple iterations of AtomTracker-I.

As AtomTracker-I runs, it can fix itself. Indeed, if later-on in the trace file, Thread 2 writes to x in between the two correct ARs as in Figure 3.3(c), AtomTracker-I will record that I_2 is the end of the AR starting at I_1. Then, in a second iteration of AtomTracker-I on the trace file (as per Section 3.4.1), as AtomTracker-I reaches I_1 in Figure 3.3(a), it will pick up the two ARs from its database — the smaller one that terminates at I_2 and the larger one that finishes at I_3. Since AtomTracker-I reaches I_2 next, it will confirm the smaller AR and discard the larger AR. The result will be the ARs of Figure 3.3(d), which are correct. This is an example of how multiple

iterations help AtomTracker-I converge to the correct ARs.

After this, AtomTracker-I moves to process another trace file. The same convergence described above may occur across trace files. When the processing of several new trace files does not result in changes in the inferred ARs, AtomTracker-I completes.

3.5 AtomTracker-D: Automatic Detection of Atomicity Violations

AtomTracker-D takes program annotations in the binary like those inserted by AtomTracker-I and automatically detects violations of ARs at *runtime*. Beyond this, AtomTracker-D is independent of AtomTracker-I.

The idea behind AtomTracker-D is as follows. As two ARs execute concurrently, AtomTracker-D checks whether they *can be made to appear to execute in sequence*, one after the other, in any of the two possible orders. Only if they cannot, AtomTracker-D declares an atomicity violation.

To do its checks, AtomTracker-D uses a novel algorithm. The algorithm considers each access of the two (or more) concurrent ARs in sequential order and checks for conflicts. The algorithm can be efficiently implemented in hardware by leveraging the cache coherence protocol messages. In this section, we describe the algorithm as it could be implemented in a tool like Pin [40], while in Section 3.6, we describe a hardware implementation.

3.5.1 Description of the Algorithm

In this description, we assume that we have two processors executing two ARs concurrently; the algorithm will be later generalized to any number of concurrent ARs. Let us call the processors P_R and P_S, and the ARs AR_R and AR_S, respectively. In the AtomTracker-D algorithm, P_R keeps a local flag called $Order_{RS}$ that tells, at any time, whether AR_R appears (so far) to execute *before* or *after* AR_S. P_S keeps a symmetrical flag called Order_{SR}.

Let us focus on Order$_{RS}$. Order$_{RS}$ is updated in three cases, as shown in Figure 3.4. Case $P_R \rightarrow P_S$ is when a reference by P_S accesses a variable that has already been referenced by P_R in AR_R. In this case, we check the following references *to the variable*: (1) the current one by P_S and (2) all of the previous ones by P_R in AR_R. If at least one is a write, then P_S is dependent on P_R and, therefore, AtomTracker-D tries to set Order$_{RS}$ to *Before*. If, instead, all of them are reads, then P_S is not dependent on P_R and AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

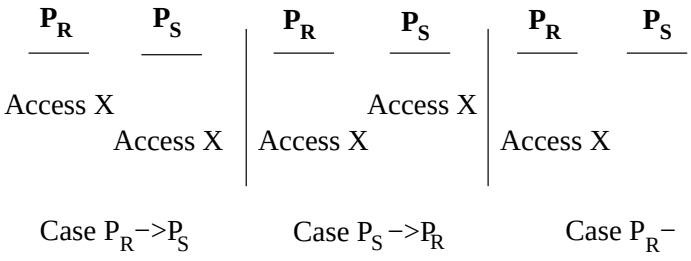


Figure 3.4: Different cases for AtomTracker-D.

The second case (Case $P_S \rightarrow P_R$) is the dual case: a reference by P_R accesses a variable that has already been referenced by P_S in AR_S. As before, we check if at least one of the relevant references to the variable is a write. If so, P_R is dependent on P_S and AtomTracker-D tries to set Order$_{RS}$ to *After*. Otherwise (i.e., the references to the variable are all reads), P_R is not dependent on P_S and AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

The final case (Case P_R-) is when P_R accesses a variable that P_S has not accessed in AR_S yet. In this case, AtomTracker-D tries to set Order$_{RS}$ to *Unordered*.

We say that the algorithm *tries* to set Order$_{RS}$ to a value because what it really does is to set Order$_{RS}$ to the logical AND of the value and the old contents of Order$_{RS}$. This is done to detect any ordering inconsistency: the AND of *Unordered* with any other value is that other value, while the AND of *Before* and *After* signals an ordering inconsistency. In this latter case, the two ARs cannot be serialized, and we have an atomicity violation.

3.5.2 Illustrative Examples

Figure 3.5 shows two examples where processors P1 and P2 execute atomic regions AR1 and AR2, respectively. It also shows the updates to Order$_{P1P2}$. We start with Figure 3.5(a). The first access (I_1) in the figure is a write to *x* by P1. This access falls into Case *P1-* because P2 has not accessed *x* in AR2 yet. Therefore, Order$_{P1P2}$ is set to *Unordered*. In access J_1, P2 reads *x*. This is Case $P_1 \rightarrow P_2$ with a write, and AtomTracker-D logically-ANDs *Before* to Order$_{P1P2}$. The result is *Before*. In access I_2, P1 reads *x*, which is Case $P_2 \rightarrow P_1$ with all reads, and AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. The result is *Before*. In access I_3, P1 reads *y*, which is Case *P1-*, and AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. The result is *Before*. Next, in access J_2, P2 writes *y*, which is Case $P_1 \rightarrow P_2$ with a write, and AtomTracker-D logically-ANDs *Before* to Order$_{P1P2}$. The result is *Before*. Next, in access I_4, P1 reads *x*, which is Case $P_2 \rightarrow P_1$ with all reads. AtomTracker-D logically-ANDs *Unordered* to Order$_{P1P2}$. This was the last access in AR1 and the algorithm concludes that there is no atomicity violation because AR1 can appear to execute *before* AR2.

Consider now Figure 3.5(b), which changes I_4 from a read to a write. Access I_4 is Case $P_2 \rightarrow P_1$ with a write. Consequently, AtomTracker-D logically-ANDs *After* to Order$_{P1P2}$, which triggers a violation. Effectively, this access requires AR1 to be *after* AR2, which is incompatible with the other accesses.

In these examples, as soon as the first processor completes its AR, the algorithm can declare the presence or absence of a violation. However, this is not always the case. Specifically, if the Order flag of the processor that finishes first has the value *After*, AtomTracker-D cannot declare the outcome until the other processor also finishes its AR.

This case is shown in Figure 3.6, which shows both Order$_{P1P2}$ and Order$_{P2P1}$. The example shows a dependence from P1 to P2, and one in the opposite direction. Consequently, there is an atomicity violation. We focus first on P1 and its flag Order$_{P1P2}$. Reference I_1 sets the flag to *Unordered*, and J_1 ANDs *Unordered* to it. Reference I_2 is a read to *y* by P1, which is Case

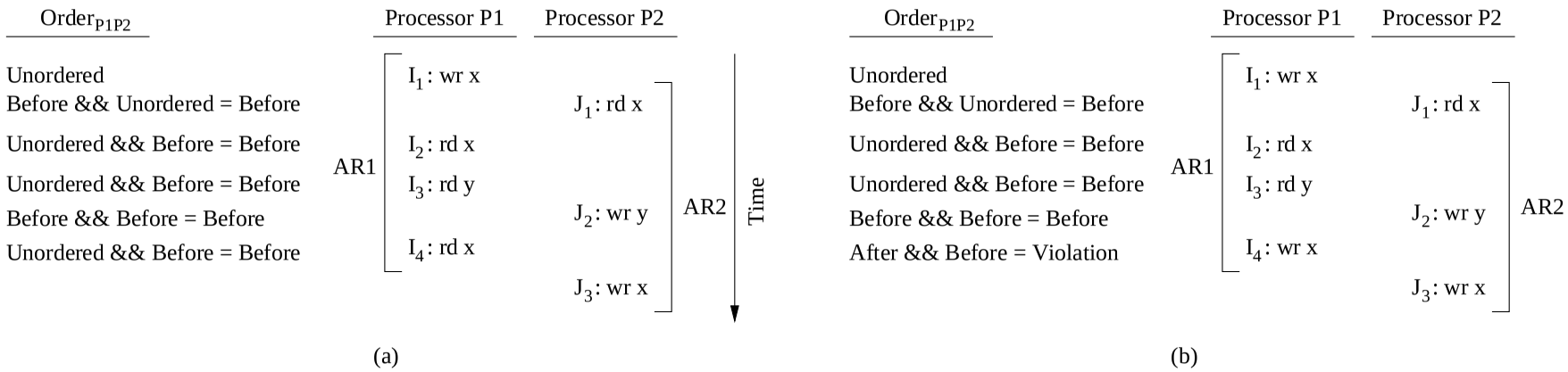


Figure 3.5: Two examples of the AtomTracker-D algorithm.

$P_2 \rightarrow P_1$ with a write. Consequently, AtomTracker-D logically-ANDs *After* to Order$_{P1P2}$. At this point, AR1 completes. However, AtomTracker-D cannot strictly declare an outcome because the AR that appears to execute first (AR2) is not complete yet.

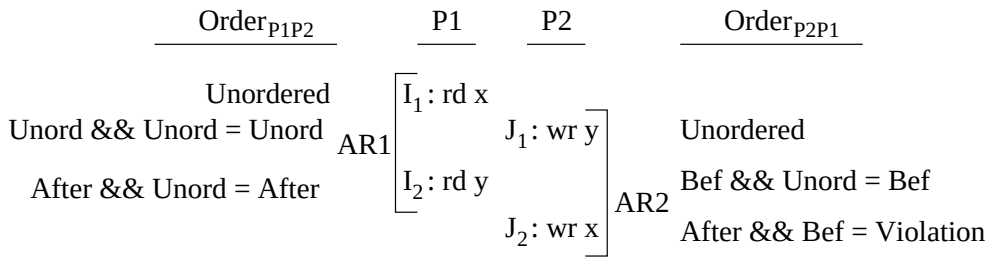


Figure 3.6: Both processors need to complete their ARs for AtomTracker-D to declare the outcome.

We now examine P2 and its flag Order$_{P2P1}$. Reference J_1 sets Order$_{P2P1}$ to *Unordered*. Reference I_2 ANDs *Before* to Order$_{P2P1}$. Finally, reference J_2 ANDs *After* to Order$_{P2P1}$. At this point, AtomTracker-D correctly declares a violation. Note, however, that the information that P1 had read *x* in AR1 *has to be kept around* until AR2 completes.

3.5.3 Generalization to More Atomic Regions

In the most general case, all *N* processors in a multicore may be executing atomic regions AR_0, $AR_1 ... AR_{N-1}$ concurrently, and AtomTracker-D has to detect atomicity violations between any two ARs. Consequently, each processor *i* keeps N-1 Order$_{i*}$ flags, where * takes the values from *0* to *N-1* except *i*.

Given a processor P_i, its Order$_{i*}$ flags are updated following the above description. Specifically, when another processor P_j accesses a variable that has been accessed by P_i, Order$_{ij}$ is ANDed with *Before* or *Unordered*. When P_i accesses a variable that has been accessed by P_j, Order$_{ij}$ is ANDed with *After* or *Unordered*. Finally, when P_i accesses a variable that P_j has not accessed yet, Order$_{ij}$ is ANDed with *Unordered*.

3.6 Hardware Implementation

Both AtomTracker-I and AtomTracker-D can be easily implemented in software. However, if we want to run AtomTracker-D on-the-fly in production runs, a software implementation is too slow. Consequently, we propose an efficient hardware implementation of AtomTracker-D.

A key insight is that the AtomTracker-D algorithm does not really need to observe every single access in the two (or more) concurrent ARs. Instead, it only needs to observe those accesses that induce cache coherence transactions in the network — with some exceptions that we will handle. Consequently, we propose to add a hardware module called *Atomicity Violation Detection Module* (AVM) attached to the on-chip network of the multicore. The AVM sees all the relevant accesses and runs the AtomTracker-D algorithm in hardware. It supports atomicity violation detection without slowing down execution.

For simplicity, our AVM design is centralized. It is possible to distribute the design to make it scalable.

3.6.1 Leveraging Cache Coherence Transactions

Many of the references processed in the AtomTracker-D algorithm of Section 3.5.1 are guaranteed not to change the value of the Order flag. For example, when a processor writes the same variable multiple times without any intervening access from other processors, then, after the first write, the value of the Order flag will not change. Consequently, it suffices that we capture only the references that *can* change the value of Order.

One group of accesses that can change Order are those that introduce RAW, WAW, or WAR dependences on a variable between two processors. These are cases $P_R \rightarrow P_S$ and $P_S \rightarrow P_R$ with a write in Figure 3.4. In particular, in a sequence of such dependences on a given variable between a given source and a given destination processor, AtomTracker-D only needs to observe one dependence. Fortunately, by construction (and ignoring false sharing for now) the first one of these dependences induces a change in the variable's cache coherence state, and a resulting

coherence transaction in the network. Consequently, it can be observed easily.

The other group of accesses that can change Order are those that introduce RAR dependences on a variable between two processors (cases $P_R \rightarrow P_S$ and $P_S \rightarrow P_R$ with only reads in Figure 3.4) or accesses to variables that have not been accessed in the other AR (case P_R- in Figure 3.4). In a sequence of such accesses to a given variable for a given source and destination processor, AtomTracker-D only needs to observe one. Again, we choose the first one. This access may miss in the cache. If so, the AVM attached to the on-chip network will see the address and process the reference. However, this access will not miss if the variable is in the cache is a certain state when the processor enters the AR. Specifically, a read that finds the variable in a state equivalent to Shared or Dirty in the cache, or a write that finds it Dirty, will not miss.

To ensure that these accesses are visible to the AVM, we modify the cache to operate slightly differently when executing an AR. We add two *FirstAccess* bits per cache line — one for reads and one for writes. Every reference in the AR sets the corresponding *FirstAccess* bit in the line touched. If the reference hits in the cache and the corresponding bit was not set, the line address (and whether the access was a read or a write) is sent to the network, so that the AVM captures it; if the reference misses in the cache, the AVM sees it by default. *FirstAccess* bits are cleared when an AR finishes.

Finally, caches naturally replace lines and references that should otherwise not miss may miss. This means that the AVM will see more accesses than the strict minimum necessary for running AtomTracker-D. This does not affect correctness in any way.

3.6.2 An Atomicity Violation Detection Module (AVM) Based on Address Signatures

A naive AVM design would have, for each processor, a buffer with the list of references since the current AR started. The references would be processed using the AtomTracker-D algorithm. However, we propose a more efficient approach based on hardware address signatures. These are

registers of about 2048 bits that accumulate the result of hashing addresses of references using Bloom filters [6]. Conceptually, a signature acts like a compressed buffer to store memory references. Per processor, the AVM has one signature (*RSig*) that hash-accumulates the addresses read in the AR and one (*WSig*) for the addresses written. Every network transaction by a processor executing an AR is captured by the AVM and the address is encoded in the correct signature. The AVM is shown in Figure 3.7. Each pair of signatures has N-1 associated Order flags.

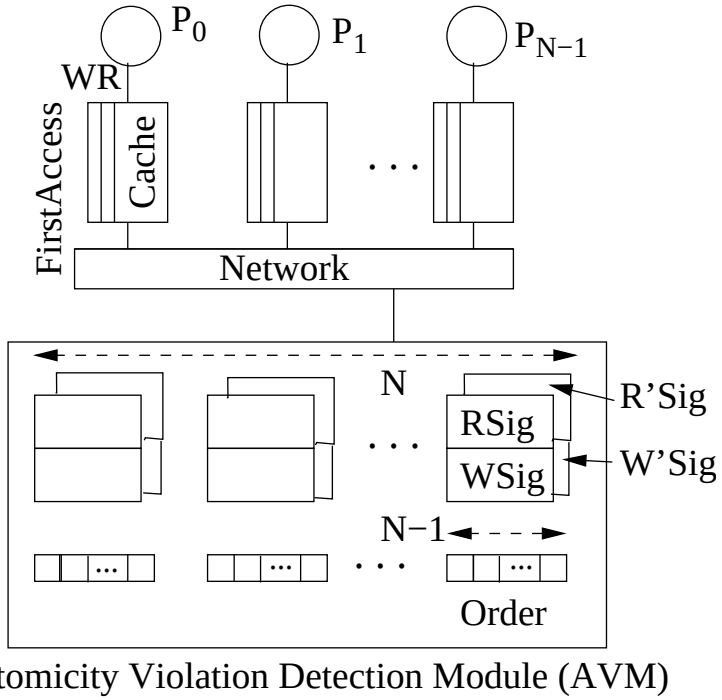


Figure 3.7: Signature-based hardware implementation of AtomTracker-D in a multicore.

AtomTracker-D is implemented as follows. Assume that the AVM observes a transaction by processor P_i to address *Addr*. The address is hashed and accumulated into the appropriate signature (RSig_i or WSig_i). Assume it is a write. In this case, the AVM hardware uses membership signature operations [13] to see if *Addr* is present in RSig_n or WSig_n, for all processors $P_n \neq P_i$. For each processor P_j where the answer is "yes", we have found a WAR or WAW. The AVM hardware does the following. Since this is Case $P_j \rightarrow P_i$ with write access, the AVM ANDs Order_{ij} with *After* and, in addition, ANDs Order_{ji} with *Before*. Finally, for each processor P_k where the answer is "no", since this is Case P_i-, the AVM ANDs Order_{ik} with *Unordered*.

If the access is a read, the procedure is similar. AtomTracker-D first checks if *Addr* is present in WSig$_n$. For each processor P_j where the answer is "yes", we have found a RAW, and the AVM hardware ANDs Order$_{ij}$ with *After* and Order$_{ji}$ with *Before*. For the other processors, AtomTracker-D checks if *Addr* is present in RSig$_n$. For each processor P_j where the answer is "yes", we have found Case $P_j \rightarrow P_i$ with RAR only, and the hardware ANDs Order$_{ij}$ and Order$_{ji}$ with *Unordered*. Finally, for the other processors P_k, this is Case P_i-, and the hardware ANDs Order$_{ik}$ with *Unordered*. These operations are done in parallel.

Signatures do not keep precise information and, as they contain more addresses, they appear to include a larger number of other addresses as well — also called aliases. This may cause false positives, an issue we address in Section 3.6.4. To minimize such events, the AVM keeps several (four in our design) physical signatures for each logical signature. Each hashed incoming address is accumulated into one of the four signatures of the logical one, *depending on its address*. The number of operations on signatures does not change. Overall, at the cost of more hardware, this design reduces address aliasing.

As soon as the AVM detects a violation, it records it. When a processor ends its AR, its signatures are cleared. However, recall from Figure 3.6 that if the Order flag is set to *After* when a processor finishes its AR, and the concurrent AR is not yet finished, we cannot discard the state until the concurrent AR ends. Consequently, in this case, the AVM saves the processor's *RSig* and *WSig* into Shadow Signatures (*R'Sig* and *W'Sig* in Figure 3.7). These shadow signatures are checked by references from the concurrent AR until the latter finishes.

3.6.3 Software Interface

The AVM is driven by two instructions that are inserted in the program by either AtomTracker-I or another software tool. They are *atomic_enter* and *atomic_exit* (Table 3.1). *Atomic_enter* marks the entry to an AR. It triggers the allocation of signatures and Order flags in the AVM. It saves the PC of *atomic_enter* in a processor register that we call *AtomTrackerEntry*. This register will be used to identify the matching *atomic_exit* instruction. In addition, *atomic_enter* sets the cache

operation to AR mode. In this mode, cache accesses set the *FirstAccess* bits of the lines touched. If the cache intercepts the access and the corresponding *FirstAccess* bit is clear, the hardware sends the line address (plus whether this is a read or a write) to the network, so that the AVM captures it. As indicated in Section 3.4.2, AR entries and exits may be unpaired and, therefore, execution may already be in an AR. In this case, *atomic_enter* has no effect.

Instruction	Description
atomic_enter	If (found outside an atomic region) Allocate signatures/flags in AVM *AtomTrackerEntry* = PC Set cache to AR mode. Per mem. access: If [(cache intercepts access) and (*FirstAccess* bit = 0)] Send line address + R/W to network *FirstAccess* bit = 1
atomic_exit PC	If [(PC = *AtomTrackerEntry*) or (NumMismatches = MAXMISMATCH)] Set cache to plain mode: *FirstAccess* bits = 0 for all cache lines Disable *FirstAccess* bit use *AtomTrackerEntry* = 0 Deallocate own structures in AVM

Table 3.1: Instructions added to manage the AVM.

Atomic_exit marks the exit of an AR. It takes the PC of the matching *atomic_enter* instruction. This instruction only has an effect if its PC argument is equal to the *AtomTrackerEntry* contents, or if we found a threshold number of mismatching *atomic_exit* instructions in a row. This is done to ensure that the current AR eventually finishes. In these cases, *atomic_exit* exits the AR by setting the cache operation to plain mode (no *FirstAccess* use), clearing *AtomTrackerEntry*, and deallocating its AVM structures.

3.6.4 Design Issues

Address aliasing in the signatures could result in false positives (FPs), namely claims of atomicity violations when there are none. The actual signature implementation affects the number of FPs.

Consequently, we have chosen the design described in [50], which has few FPs. On the other hand, signatures cannot lead to false negatives, namely false claims of no atomicity violations.

Network accesses only expose *line* addresses. Therefore, all AtomTracker-D operations are done at cache-line granularity. This makes the implementation subject to false sharing, which can also result in false positive claims, but not false negatives.

The *atomic_enter* and *atomic_exit* instructions use fences, as if they were synchronizations. Therefore, the AtomTracker-D hardware also works with relaxed memory consistency model machines.

3.7 Conclusions

Our proposal AtomTrackeris the *first* scheme to (1) automatically infer *generic* non-nested ARs (not limited by issues such as the number of variables accessed, the number of instructions included, or the type of code construct the region is embedded in) and (2) automatically detect violations of them at runtime with negligible execution overhead. No programmer input or annotations are needed. AtomTracker provides novel algorithms to infer generic ARs and to detect atomicity violations of them. Moreover, we presented a hardware implementation of the violation detection algorithm that leverages cache coherence state transitions in a multiprocessor. To evaluate AtomTracker, we took eight atomicity violation bugs from real-world codes like Apache, MySql, and Mozilla, and showed that AtomTracker detects them all — which is not the case with any of the existing approaches. In addition,

Chapter 4

Detecting Sequential Consistency Violations

4.1 Introduction

The model that programmers have in mind when they program — and debug — shared-memory threads is Sequential Consistency (SC). SC requires the memory operations of a program to appear to have executed in some global sequential order that is consistent with each thread's program order [28]. In practice, however, current processor hardware aggressively overlaps, pipelines, and reorders the memory accesses of a thread. As a result, a program's execution can be very unintuitive.

As an example, consider the simple case of Figure 4.1(a). In the example, processor P_A allocates a variable and then sets a flag. Later, P_B tests the flag and uses the variable. While this interleaving produced expected results, the interleaving in Figure 4.1(b) did not. In here, since the variable and the flag have no data dependence, the P_A hardware reorders the statements. In this unlucky interleaving, P_B ends up using uninitialized data. This is an SC violation.

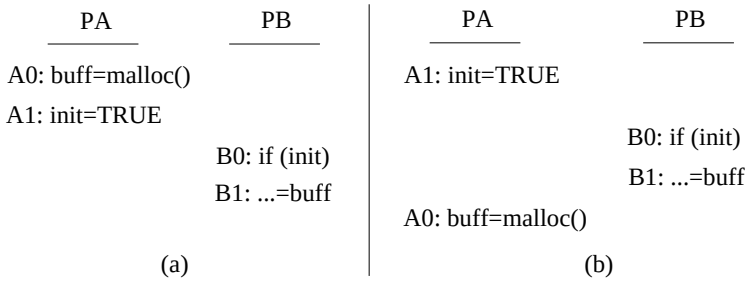


Figure 4.1: Example of an SC violation.

For an SC violation to occur, it is necessary to have data races. Consequently, if the program is

well synchronized (i.e., race-free), then the memory access reorderings induced by the hardware do not matter. However, in practice, codes often contain data races. Such races may be inserted inadvertently by the programmer, or created as a side effect of a careless compiler transformation, or even introduced on purpose by the programmer — typically to improve the performance of the code. Common intentional data races are found in double-checked locking constructs [66], some synchronization libraries, and code that implements synchronization-free data structures.

While past work has used race-detection techniques to find SC violations [24, 38, 43], using data races as proxies for SC violations is very unproductive. One reason is that a large majority of data races in the codes are of the so-called *benign* kind [19, 52]. These races are informally defined as those that, no matter the reference interleaving, do not change the program output (in a way that the programmer cares about).

A second reason is that, even among *harmful* data races, an SC violation requires a very special type of race. Indeed, it requires the presence of at least *two overlapping* data races *intertwined* in a very unique manner [67]. Specifically, for two threads, it requires a pattern like in Figure 4.2(a) where, if we follow program order, the two threads reference the same two variables in opposite orders, and each variable is written at least once. Moreover, the order of the dependences between these two racing pairs *has to form a cycle* at runtime. This is shown in Figure 4.2(b), where we have arbitrarily assigned reads and writes: *A1* must occur before *B0* and *B1* must occur before *A0*. This is exactly what happened in Figure 4.1(b), where *y* was *init* and *x* was *buff*.

Note, however, that if the timing is such that at least one of the two dependence arrows occurs in the opposite direction at runtime, there is no SC violation. For example, Figure 4.2(c) shows the case when *A0* executed before *B1*. Since there is no cycle, SC is not violated. This case corresponds to the timing in Figure 4.1(a). The other parts of Figure 4.2 are described later.

A final reason for not using data races as proxies is that, in many cases, we want to find SC violations in codes that have intentional data races, such as synchronization-free data structure codes. We want to debug the code for SC violations while retaining the non-SC-violating races. Here, a race-detection tool would be the wrong instrument and create many false positives.

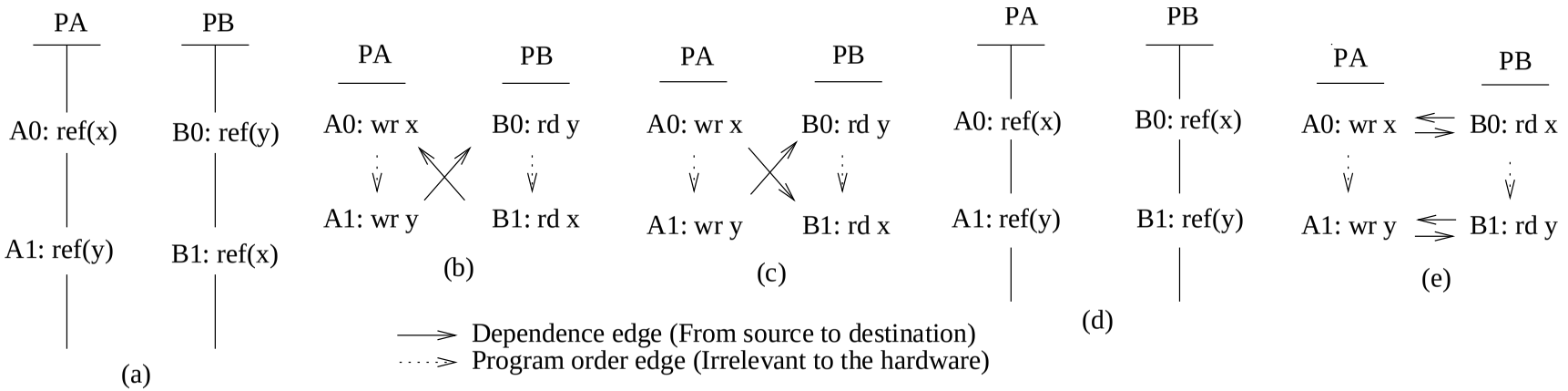


Figure 4.2: Understanding SC violations.

From this discussion, it is clear that we need a technique that *precisely* zeroes-in on these clusters of data races and interleavings. Developing a precise, sophisticated technique for it is justified because virtually all SC violations are harmful races and clear-cut bugs. The reason is that, as the example in Figure 4.1(b) showed, they *require* memory access orders that contradict a programmer's intuition. In addition, they are doubly harmful because the programmer cannot reproduce them using a single-stepping debugger.

4.2 Contributions

Given the importance of these bugs and the difficulty in isolating them, this work contributes with *Vulcan*, the first hardware scheme to dynamically detect SC violations in a program running on a relaxed-consistency machine *precisely*. Vulcan's idea is to leverage cache coherence protocol transactions to dynamically detect cycles in memory access orderings across threads. When one such cycle is about to occur, an exception is triggered, providing the exact architectural state. Vulcan's approach has several advantages: it suffers from no false positives or false negatives, induces negligible execution overhead, requires no help from the compiler, and only takes as input the program executable.

Experimental results show that Vulcan detects *three new bugs* in very popular codes. Specifically, Vulcan finds SC violations in the Pthread and Crypt libraries, and in the fmm program from SPLASH-2. We have reported the bugs to the developers. In addition, Vulcan's negligible execution overhead makes it suitable for on-the-fly use in production runs.

Finally, we also contribute with a new taxonomy of data races.

4.3 Background

A Sequential Consistency Violation (SCV) occurs when the memory operations of a program have executed in an order that does not conform to any SC execution. It is virtually always a harmful

bug, since it is the outcome of an execution not predicted by the programmer and, furthermore, is the hardest to debug because debuggers cannot reproduce it.

Shasha and Snir [67] show what causes an SC violation: overlapping data races where, at runtime, the dependences end up ordered in a cycle. Recall that a data race occurs when two threads access the same memory location without an intervening synchronization and at least one is writing. Figure 4.2(a) showed the required program pattern for two threads (where each variable is written at least once) and Figure 4.2(b) showed the required order of the dependences observed at runtime for SCV (where we assigned reads and writes to the references arbitrarily).

If at least one of the dependences occurred in the opposite direction (e.g., Figure 4.2(c)), no SCV occurs. In addition, if the code of the two threads references the two variables in the same order (Figure 4.2(d)), no SCV is possible — no matter how these references end up being reordered at runtime. For example, in Figure 4.2(e), no SCV can occur — no matter the execution order of the instructions within a thread, or the direction of the inter-thread dependences.

Given the pattern in Figure 4.2(a), Shasha and Snir [67] avoid the SCV by placing a fence between the references *A0* and *A1* and another between the references *B0* and *B1*. Their algorithm to find where to put the fences has been called the Delay Set.

The commonly used Double-Checked Locking (DCL) [66] is a major source of SCVs. This is a programming technique to reduce the overhead of acquiring a lock by first testing the locking criterion without actually acquiring the lock. Only if the test indicates that locking is required does the actual locking logic proceed. The code takes a structure like that in Figure 4.1(a). Because the code is typically involved, programmers often miss putting the two fences required by Shasha and Snir. In that case, an SCV may occur.

Data races and SCVs are very different and, intuitively, programs have many more data races than SCVs. However, past work has focused on detecting data races as proxies for SCVs. This includes the work of Gharachorloo and Gibbons [24], DRFx [43] and Conflict Exceptions [38]. Specifically, they all look for a data race between two accesses from different threads that occur within a short time distance. The actual detection scheme is different: Gharachorloo and Gibbons

detect external invalidations (or read requests) on local outstanding loads and stores, while DRFx and Conflict Exceptions detect conflicts between concurrent synchronization-free regions. Moreover, when such a race is detected, DRFx and Conflict Exceptions raise an exception. Overall, while focusing on a data race between close-proximity references may be a good way to discard many irrelevant races, it is still a very different problem than focusing on uncovering SCVs.

Other researchers have used the compiler to identify race pairs that could cause SCVs, typically using the Delay Set algorithm, and then insert fences to prevent cycles [18, 20, 27, 30, 31, 70]. In general, compiler approaches tend to be very conservative, and typically result in substantial slowdowns. There is some work on program testing and verification that either checks semantic correctness or collects traces of a program and then, off-line, applies reordering rules to try to detect SCVs [8, 9, 10]. While such techniques are promising, they are typically limited to small-sized codes and are performed statically or as an off-line pass. Finally, the hardware verification community tries to verify whether the memory subsystem hardware correctly implements SC or some other memory consistency models [14, 17, 44]. Our work is different in that we focus on debugging software as it runs on a correctly-implemented relaxed-consistency machine. More details on related work are presented in Section 4.4.

4.4 Related Work

Beyond the work of Gharachorloo and Gibbons [24], DRFx [43] and Conflict Exceptions [38] already described in Section 4.3, the most related work has to do with compiler-inserted fences, testing, and hardware verification.

The compiler community has done substantial work on avoiding SCVs by inserting fences (e.g., [20, 30, 70]). Most of the work is based on the Delay Set algorithm [67]. Krishnamurthy and Yelick [27] reduce the complexity of the algorithm by applying it to SPMD programs. Sura et al. [70] combine delay set analysis with escape and thread structure analysis to make it more effective. Duan et al. [18] combine delay set analysis with a dynamic data race detector to improve

the precision of inserting fences.

Lin et al. [31] reduce the overhead of the SC-enforcing fences by proposing a conditional fence (C-Fence). They point out that, for an SCV to occur, the cross-thread dependence arrows must form a cycle (Figure 4.2(b)). If one arrow points in the opposite direction because the source thread has executed long ago, then an SCV violation is impossible and there is no need for the second fence. They propose a fence that, in such conditions, becomes invisible.

The testing community has proposed static and off-line techniques to check for SCVs. For example, Burnim et al. [10] and SOBER [9] collect SC traces of a program. Then, they apply the reordering rules of various memory models on them, hoping to detect SCVs. CheckFence [8] checks semantic consistency by enumerating all possible executions and using an SAT solver.

There are some works from the hardware verification community that try to verify if a particular memory subsystem hardware has been correctly implemented. While somewhat related, these works have a different goal than ours: we assume correct hardware and check if the software is correct; they check if the hardware is correct and do not care about the software. For example, Meixner and Sorin [44] verify that the memory subsystem hardware supports SC either directly by checking some memory ordering rules or indirectly by checking some invariants. They induce some hardware errors and then show that they can detect them. Chen et al. [14] verify different memory models by building constraint graphs and looking for cycles. They use various techniques to reduce the size of the graph. DACOTA [17] collects hardware activity logs and periodically examines the logs to verify that the hardware was SC. Our work is different in that we focus on debugging software as it runs on a relaxed-consistency machine; their focus is on verifying that the memory system hardware correctly implements a memory model.

4.5 A New Taxonomy of Data Races

To assess the relationship between data races and SCVs, we develop a new taxonomy of data races. We examined the bug databases of popular programs such as Apache, MySQL, and Mozilla, and

App.	# Harmful Races	Bug ID	# Multi-Races	# SCV Races	# DCL SCVs
Apache	24	254653, 49972, 25520, 21287, 49985, 49986, 47158, 48790, 31018, 45608, 40681, 40167, 37458, 36594, 37529, 40170, 44402, 44178, 46215, 41659, 45605, 1507, 46211, 728	5	5	5
MySQL	13	644, 791, 2011, 3596, 12848, 19938, 24721, 48930, 42101, 45058, 56324, 52356, 59464	1	1	1
Mozilla	11	622691, 341323, 133773, 342577, 52111, 224911, 270689, 73761, 124923 225525, 325521	2	1	1
Redhat (glibc)	**2**	2644, 11449	**2**	**2**	**1**
Java SDK	2	6633229, race in ConcurrentLinkedQueue	1	1	1
PostgreSQL	1	613	0	0	0
Pbzip2	1	Data race from [76]	0	0	0
Windows kernel	1	Data race from [19]	0	0	0
Isolator bench.	1	Data race from Isolator [61]	0	0	0
SPLASH-2	**1**	Data race from fmm	**1**	**1**	0
Total	57	—-	12	11	9

Table 4.1: Harmful data races studied. The races in bold are *three new SC Violation (SCV) bugs* found by Vulcan.

collected all the data-race bugs we could find. Since these are bugs reported by users, we declare them *Harmful* data races. Table 4.1 lists the applications and the number of harmful races, together with the bug IDs when available. The races in bold in the table are *three new SCV bugs* that we uncovered as we evaluated Vulcan— two of them in the official fixes of the reported Redhat bugs and one in a SPLASH-2 code. More details are presented in Section 5.3.3.

Overall, we recorded 57 harmful races. For each of these bugs, if they contain more than one race, we call them *Multi-races*; otherwise, we call them *Single-races*. In addition, if a multi-race bug can create an SCV, we call it an *SCV Race*; otherwise it is a *Non-SCV Race*. Finally, SCVs are classified into those that are DCLs [66] and those that are not.

Table 4.1 shows the breakdown of the bugs per application. We see that, of the total 57 harmful races, 12 are multi-races (21%). This is a sizable fraction. Of these, 11 can cause SCVs (91%). The only one that, due to its reference pattern cannot ever create an SCV is in Mozilla [2]. Of the 11 SCVs, 9 are DCLs (82%). The ones that are not DCLs are shown later in Figures 5.6 and 5.8.

To put this in context, Microsoft reported that about 90% of all the data races are benign [19, 52]. Therefore, we can build the tree of Figure 4.3(a), which shows the frequency of each type of bug relative to its parent's. To visualize the frequency relative to all the race instances, Figure 4.3(b) shows a diagram where the area is proportional to the frequency of occurrence. We can see that SCVs account for 1.9% of all the data races. This suggests that previous approaches that focus on data races as proxies for SCVs [24, 38, 43] are insufficient.

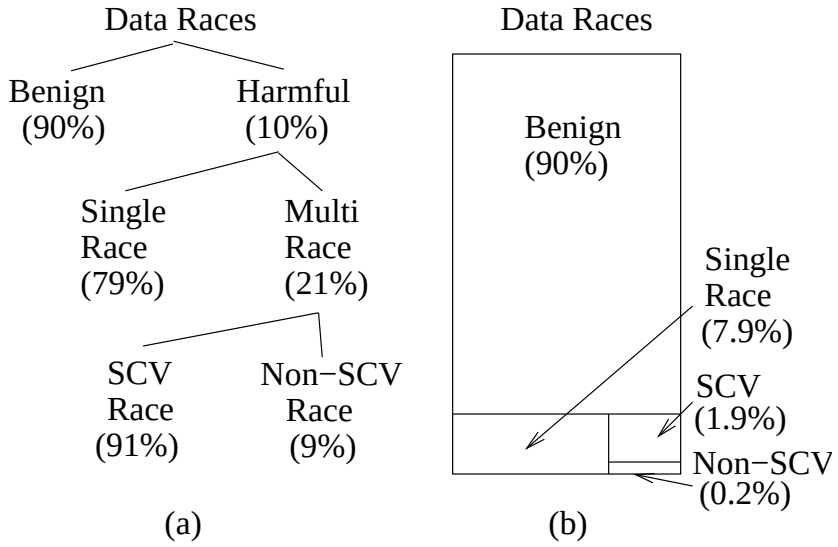


Figure 4.3: Relative frequency of data-race types.

The figure also shows why a special technique for SCV bugs is warranted: they comprise a substantial fraction of the harmful data races, namely 0.91x21%=19.1%. In addition, they are very hard to debug, since debuggers cannot reproduce them. Finally, there are no existing precise techniques to find them. For example, even a sophisticated data-race detector is ineffective, since we often look for SCVs in the codes that have intentional data races, such as libraries of synchronization-free data structures. Using a race detector would continuously trigger false posi-

tives.

4.6 Vulcan: Detecting SC Violations

Our goal is to develop a *precise* approach to detect SCVs in relaxed-consistency machines. Hence, we focus on detecting cycles in inter-thread data dependences at runtime. In addition, we want a solution that has no false positives or false negatives, is able to deliver the exact processor state with an exception at an SCV, and has a negligible execution overhead. Finally, the solution should not rely on the compiler because compilers are conservative — it should take a plain executable.

The idea behind our approach, called *Vulcan*, is to rely on the cache coherence protocol to dynamically record the observed inter-thread data dependences, and then check whether they form cycles. These dependences are kept around only for as long as they can participate in a cycle, and are discarded soon after. Both the recording and the checking of these dependences is done in hardware for minimum execution overhead. In this section, we describe Vulcan.

4.6.1 Basic Algorithm to Detect Cycles

Figure 4.4(a) repeats the pattern that can lead to an SCV with two threads. An SCV occurs when, due to the out-of-order execution of *ref(x)* and *ref(y)* in one thread or in both threads, *A1* executes before *B0*, and *B1* executes before *A0* — creating a dependence cycle.

In Vulcan, the hardware dynamically captures the dependences and checks for cycles. To understand how it works, consider the arrow of Figure 4.4(b), which represents that reference *A1* executed before reference *B0*. This arrow creates two regions, *R1* and *R2*, such that any future dependence whose source is in *R1* and destination is in *R2* will cause an SC violation. Consequently, after Vulcan records *A1*→*B0*, it monitors that no new dependence is created from an access in P_B at or after *B0* to an access in P_A at or before *A1*.

We put this requirement as the two restrictions of Figure 4.4(c):

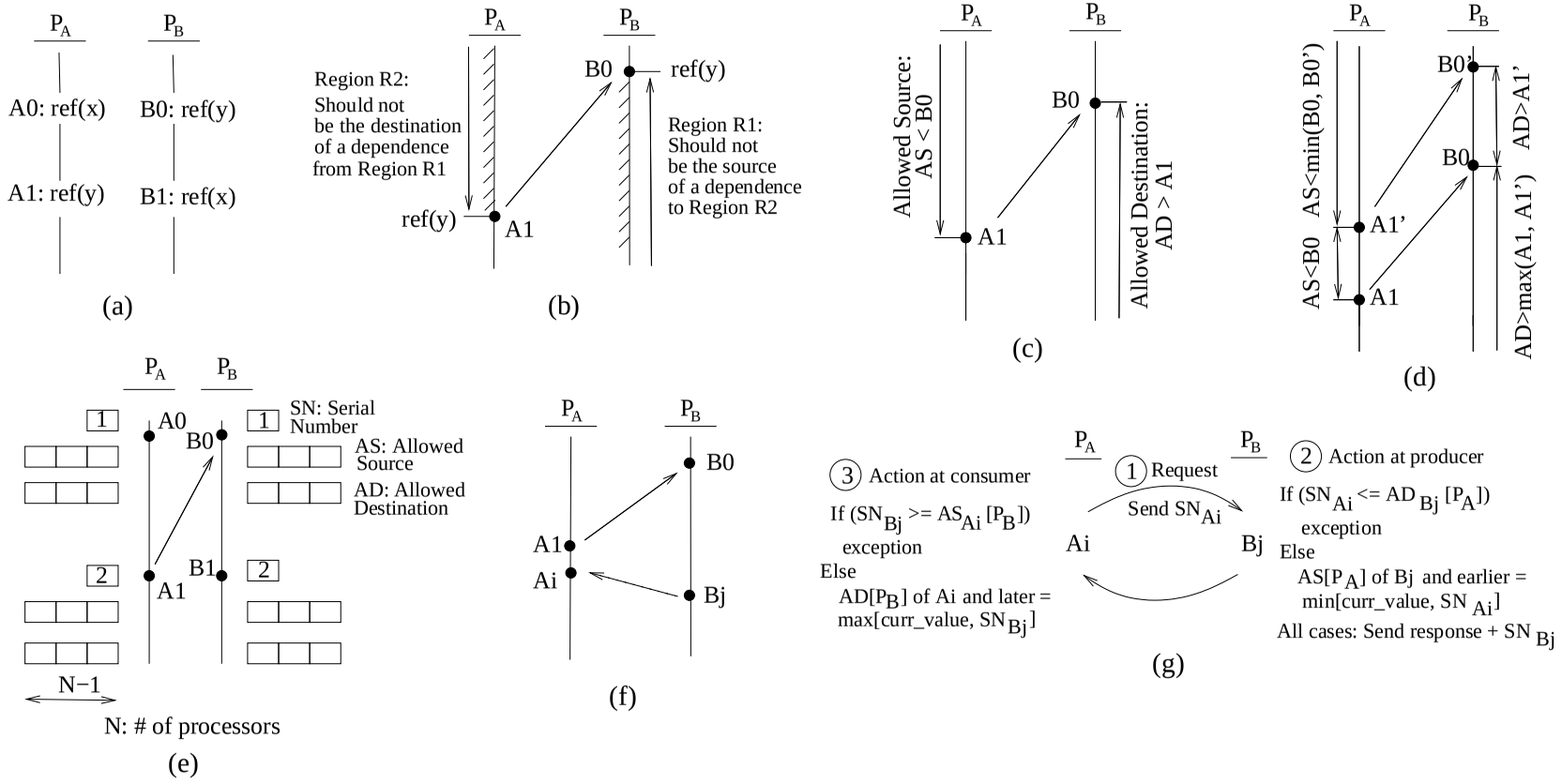


Figure 4.4: Basic algorithm to detect cycles.

• For any dependence whose source reference is in P_B at or after *B0*, the Allowed Destination (AD) in P_A is after *A1*.

• For any dependence whose destination reference is in P_A at or before *A1*, the Allowed Source (AS) in P_B is before *B0*.

If there are multiple dependences between two threads, then the *AD* of a dependence *from* a reference is the latest (i.e., maximum) of the contributing *AD*s, while the *AS* of a dependence *to* a reference is the earliest (i.e., minimum) of the contributing *AS*s. This is shown in Figure 4.4(d). In the figure, for each of the two dependences, we use the algorithm of Figure 4.4(c) to set the *AD*s of their R1 Regions and the *AS*s of their R2 regions. In the areas where the two R1 regions overlap (*B0* and later in P_B), Vulcan sets the *AD* to the maximum of the two values; in the areas where the two R2 regions overlap (*A1'* and earlier in P_A), Vulcan sets the *AS* to the minimum of the two values.

Based on this discussion, Vulcan tags each monitored reference with three labels. They are shown in Figure 4.4(e). The first one is the Serial Number (SN), which is the local dynamic reference count, assigned when the load or store enters the pipeline (e.g., at issue). The second one is the Allowed Destination (AD), which is the *SN* of the reference in the other processor after which the local reference can send data to. The last one is the Allowed Source (AS), which is the *SN* of the reference in the other processor before which the local reference can receive data from. Since a processor can have dependences with every other processor, *AD* and *AS* are arrays of *N*-1 entries, where *N* is the processor count. In each processor, *SN* starts up as 0 and increases monotonically. *AD* starts up as 0 and AS as ∞.

These structures are updated in hardware every time that a new cross-processor dependence is created. The algorithm is shown in Figure 4.4(g), which refers to the example in Figure 4.4(f). Assume that we already have the solid arrow *A1*→*B0*; now P_A issues a request from reference *Ai* that prompts reference *Bj* in P_B to respond, creating the dotted arrow *Bj*→*Ai*. Figure 4.4(g) shows that there are three steps. Step 1 is the request from P_A, which carries the SN of the requesting

access (SN_{Ai}). In Step 2, P_B operates on its Vulcan metadata, sends the response, and possibly raises an exception. Specifically, P_B checks that a cycle is not about to form by confirming that *Ai* is an allowed destination of *Bj*. If it is not ($SN_{Ai} \leq AD_{Bj}[P_A]$), a cycle is about to form and, therefore an SC violation is detected. In this case, P_B sends the response with the SN of the producer access (SN_{Bj}) and raises an exception. Otherwise, as in the example, the metadata is updated: the $AS[P_A]$ of *Bj* and earlier accesses in P_B are set to the minimum of their current values and SN_{Ai}. Also, P_B sends the response with SN_{Bj}.

Finally, in Step 3, when the data reaches P_A, P_A operates on its metadata and possibly raises an exception. Specifically, P_A checks that a cycle is not formed by confirming that *Bj* is an allowed source of *Ai*. If it is not ($SN_{Bj} \geq AS_{Ai}[P_B]$), a cycle is formed and an SC violation has occurred. Consequently, a precise exception is raised. Otherwise, as in the example, the $AD[P_B]$ of *Ai* and later accesses in P_A are set to the maximum of their current values and SN_{Bj}.

With this algorithm, Vulcan raises exceptions immediately when a dependence closes a cycle and an SCV occurs. This provides valuable information for debugging. In particular, the exception at the processor that receives the response provides the precise architectural state of the consumer reference closing the cycle. In addition, the exception at the processor that sends the response provides additional information from the producer reference's metadata, that can further help debug. Note that the exception at the producer may not occur since, at send time, there may not be enough dependences for a cycle yet. Still, the exception at the consumer is the one that matters and it always occurs.

4.6.2 Safe Accesses

As a processor issues references, Vulcan monitors them. To understand for how long they need to be monitored, we define the concept of a *Safe* access:

• An access is *Safe* when no data dependence involving this access can cause an SC violation any more. Vulcan can stop monitoring an access when it becomes safe.

To find out when an access becomes safe, consider the *Performed Point* (PP) in an out-of-order processor. The PP is the latest memory access (in program order) such that it and all the accesses preceding it in program order have been performed. We say that an access has been performed when any activity in the memory system resulting from it has ceased. Ignoring misspeculations, a load has been performed when the value has been loaded into the load buffer or, if forwarded, when the producer store has been performed; a store has been performed when the cache has received the data and all the invalidation acknowledgments.

As a thread executes, the PP keeps advancing. When the PP reaches an access, it is clear that the access is complete. However, the access *may still* participate in an SCV and, therefore, not be Safe. To see why, consider Figure 4.5(a). The creation of the *A1*→*B1* dependence makes the *B1* and subsequent accesses in P_B vulnerable. Indeed, even if they complete and P_B's PP goes past them, they can still participate in cycles. Specifically, if any access in P_A prior to *A1* requests data from them (or generally becomes dependent on them), a cycle is created. In precise terms: *B1* and subsequent accesses in P_B remain not Safe (i.e., *unsafe*) for as long as P_A's PP has not reached the reference in their AD (*A1* in the example).

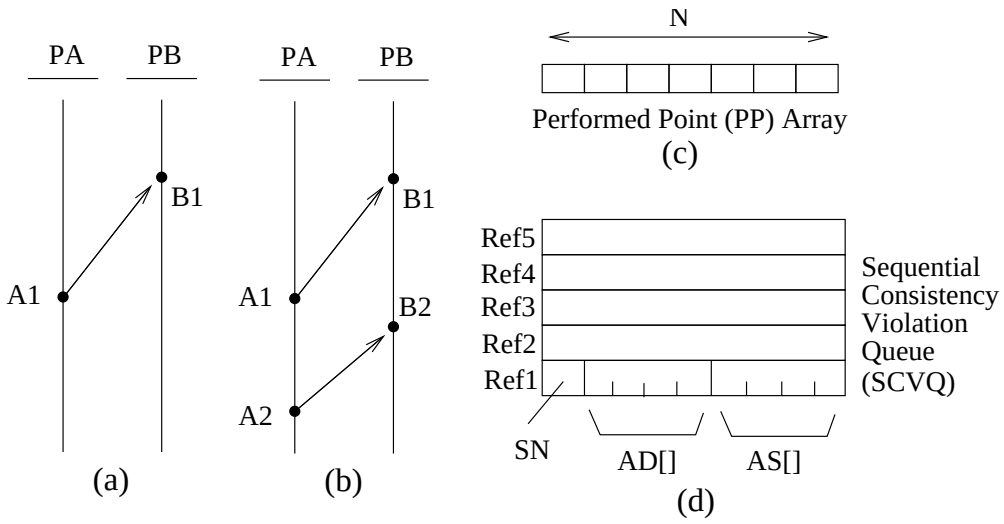


Figure 4.5: Understanding when an access is Safe.

The exact condition for an access *Ci* in processor P_C to be Safe is as follows:

• Suppose that we have an array *PP[]* with the current value of the PPs for each processor (given as *SN* numbers). *Ci* is Safe when ($SN_{Ci} \leq PP[P_C]$) and ($AD_{Ci}[P_K] \leq PP[P_K]$), for all processors $K \neq C$. [Proof in *Theorem 1* of the Appendix].

As an example, consider Figure 4.5(b). The accesses in P_A become safe as soon as $PP[P_A]$ reaches them (since their AD has not been changed from 0). The accesses in P_B remain unsafe even as $PP[P_B]$ reaches them. However, as soon as *A1* becomes safe, all the accesses in P_B up to (but not including) *B2* can become safe.

We also say that an access *Ci* in processor P_C is safe *with respect to* another processor P_M:

• *Ci* is Safe with respect to P_M when ($SN_{Ci} \leq PP[P_C]$) and ($AD_{Ci}[P_M] \leq PP[P_M]$).

Vulcan uses these insights as follows. First, each processor has a PP[] array (Figure 4.5(c)). In this array, the entries corresponding to the other processors are kept largely up-to-date thanks to the fact that each processor includes its PP in every response message.

Second, Vulcan only keeps the *SN*, *AD*, and *AS* information for the references that are not safe. Such information is kept in a per-processor FIFO hardware queue associated with the cache controller called *SC Violation Queue* (SCVQ) (Figure 4.5(d)). When the processor issues a load or store, Vulcan allocates an SCVQ entry and sets its *SN* field. Later, as the access executes and coherence actions are received, the *AD* and *AS* fields are updated. Finally, when the access becomes safe, Vulcan deallocates the entry.

An SCVQ entry does not contain the data loaded or stored. Moreover, the entry can remain allocated long after the access has completed — for as long as it remains unsafe.

4.6.3 Detecting Dependences

When an SC violation occurs, the following must be true:

• The two inter-processor dependence arrows that form the cycle must share a property: their source reference is *unsafe* with respect to the destination processor. If one of the arrows fails this condition, there is no SC violation. [Proof in *Theorem 2* of the Appendix].

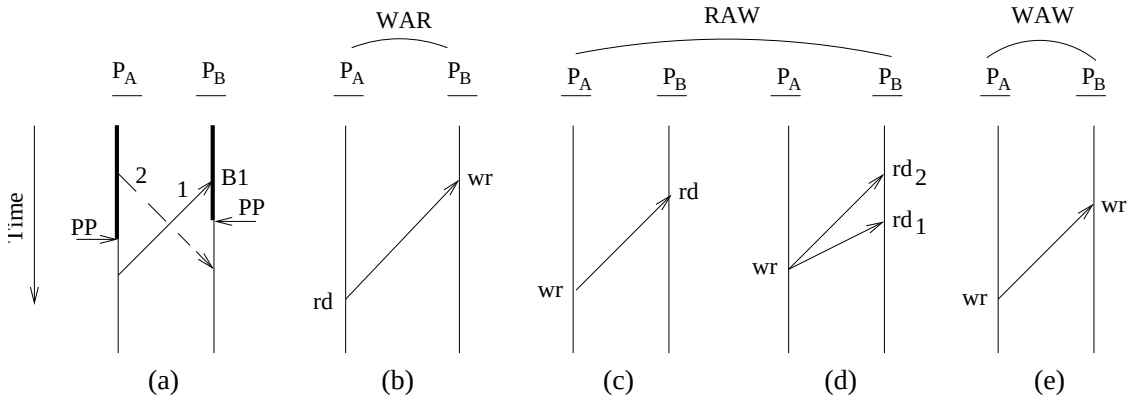


Figure 4.6: Inter-processor data dependences.

For example, in Figure 4.6(a), while arrow 1 satisfies the condition, arrow 2 does not, and no SC violation occurs. Consequently, we conclude:

• Vulcan only needs to watch for inter-processor data dependences where the source reference is unsafe with respect to the destination processor. We call such dependences *unsafe* dependences.

To find the unsafe dependences, we will see that Vulcan uses the cache coherence protocol transactions (to a large extent). When one is found, the hardware performs the basic algorithm described in Section 4.6.1: the source and destination references exchange *SN*s, the source checks its *AD* and potentially updates its *AS* (and those of earlier accesses), and the destination checks its *AS* and potentially updates its *AD* (and those of later accesses).

Figures 4.6(b)-(e) show the three types of dependences possible: WAR, RAW, and WAW. Figure 4.6(b) shows a WAR. The write triggers Vulcan to search the other processors' SCVQs for reads to the address. Multiple reader processors may be identified. Each reader processor has to take-in the write's *SN*, provide its read's *SN* and run the Vulcan algorithm; the writer has to take-in all the reads' *SN*s and run the Vulcan algorithm using the correct entries in its *AD* and *AS* arrays. In addition, since the write will be the source of the future dependence(s) on this address, the write also triggers the removal (i.e., invalidation) of the SCVQ entries for this address in all the other processors.

Figure 4.6(c) shows a RAW. The read triggers Vulcan to search the other processors' SCVQs

for a write to the address, ignoring SCVQ entries for reads. The usual algorithm is then run. Figure 4.6(d) shows a special case of a RAW, where the reader thread performs two reads to the same address *out of order*: first a later read (rd_1) and then a read that is earlier in program order (rd_2). In this case, both reads must communicate with the write's SCVQ entry. In the process, rd_1 will first set the *AS* of the write (and prior accesses) to rd_1's *SN*; later, rd_2 will set it to rd_2's *SN*, which is lower.

Figure 4.6(e) shows a WAW. As usual, the consumer write invalidates the SCVQ entry of the producer write. Note that other processors may have read the address in between the two writes. In this case, the consumer writer generates WAR dependences with the readers and a WAW dependence with the producer writer, and invalidates all the SCVQ entries for this address but its own.

Our goal is to detect all the unsafe dependences. The Appendix shows that:

• If Vulcan records all the unsafe dependences, then it can detect all the SCVs between the processors. [Proof in *Theorem 3* of the Appendix].

4.6.4 Leveraging the Coherence Protocol

To detect all the unsafe dependences,

Vulcan partially relies on piggybacking on the cache coherence protocol transactions. In this work, we describe the operation assuming a snoopy-based MSI protocol; other protocols may require slightly different arrangements. Moreover, we assume a single-level private cache hierarchy per processor, where the SCVQ is associated with the cache controller. In addition, without loss of generality, in our discussion, we use a word (i.e., 32 bits) as the finest grain of processor accesses.

To understand how Vulcan uses the coherence protocol, this section starts by assuming single-word cache lines; Section 4.7 shows the final Vulcan design, which uses multi-word lines. With single-word lines, the destination access of the WAR, RAW, and WAW dependences of Figure 4.6 induces a coherence transaction on the bus. Vulcan leverages such a transaction. The only ex-

ception is the second read (rd_2) in the RAW with out-of-order reads to the same address (Figure 4.6(d)). We describe this special case later.

As part of the coherence transaction, the Vulcan metadata is exchanged and operated upon. Specifically, on a read coherence transaction on the bus, the SCVQs that may have the referenced address (we will see later how we know this) are searched. The search tries to find the latest write to the address in program order in the SCVQ. If a write is found, we have detected a RAW. Then, the Vulcan metadata is exchanged (as part of the transaction) and operated upon appropriately.

On a write coherence transaction on the bus, the SCVQs that may have the referenced address are searched. The search tries to find the latest read or write to the address in program order in the SCVQ. Vulcan looks for the latest one because it forms the most conservative dependence. If we find one, we have detected a WAR or a WAW. Then, the metadata is exchanged and operated upon. In addition, as indicated above, all the entries for the address in the SCVQ are invalidated.

The second read (rd_2) in the RAW with out-of-order reads of Figure 4.6(d) presents a difficulty. On the one hand, the read hits in the cache and would not cause a coherence transaction. On the other hand, it needs to exchange *SN*s with the write and update the metadata (importantly, the *AS* of the write and prior accesses in P_A must become smaller). Vulcan solves the problem by forcing a *Metadata Bus Access*, namely one where only Vulcan metadata is exchanged and operated on; no data is returned. Specifically, when a load executes and finds that a later load to the same address has already executed, the hardware forces a metadata network access.

Vulcan's operation requires that, on a bus transaction, the hardware looks-up the SCVQs that may have the referenced address. Vulcan cannot rely on the cache snoopers to flag which SCVQs may have the address — since the corresponding cache line may have been evicted from the cache and have to be brought in from memory. Consequently, Vulcan adds a per-processor bloom filter that contains the current addresses in the local SCVQ. If the address on the bus hits in the filter, the SCVQ is searched. Section 4.7.3 presents a detailed design.

4.7 Vulcan Hardware Design

This section presents the key Vulcan hardware structures, namely the coherence protocol and SCVQs.

4.7.1 Supporting Multiple Words per Line

Detecting all the unsafe dependences was easy with single-word cache lines because, conveniently, in all cross-thread dependences (except RAWs with out-of-order read-read) the destination reference induces a coherence action in an MSI protocol (Figure 4.6). During the resulting bus access, the hardware exchanges and updates Vulcan metadata.

Unfortunately, this is not the case with multi-word cache lines. As a processor references a word, other words come along. Consequently, some unsafe dependences do not trigger coherence actions. Further, some coherence actions are caused by false sharing rather than by true dependences.

To solve this problem, Vulcan decouples, to some extent, the coherence actions from the Vulcan metadata operations. It ensures that every time an unsafe dependence occurs, (1) either the coherence protocol triggers a coherence action or (2) Vulcan forces a Metadata bus access.

We now use an *line-based* cache coherence protocol. However, a bus transaction also includes the address bits of the word accessed within the line. Moreover, in each valid line in the cache, Vulcan keeps two state bits per word. These bits represent the *Vulcan-State* (or V-State) of the word. A word can be in one of three V-states:

• *Solo*: This word is not in any other processor's SCVQ because the local processor has written the word and invalidated the address from all the other SCVQs, and no other processor has accessed the word since.

• *Recorded*: The local processor has recorded any unsafe dependences it may have with other processors on this word. The local processor may or may not have accessed the word.

• *Unknown*: The local processor is not up-to-date in recording unsafe dependences on this word.

These states are used as follows:

• *Solo*: The processor can read or write a *Solo* word without checking the Vulcan metadata. The access may or may not trigger a coherence action on the line in the bus. The Vulcan hardware is not exercised.

• *Recorded*: The processor can read the word without checking the Vulcan metadata. If the processor writes the word, the bus is accessed, the Vulcan metadata is operated on, and the V-state transitions to *Solo*. If the write triggers a coherence action in the bus, these actions are piggybacked on it.

• *Unknown*: If the processor reads or writes the word, it causes a bus access and operation on the Vulcan metadata. The word transitions to *Recorded* on a read or to *Solo* on a write. If the write triggers a coherence action, these actions are piggybacked on it.

We handle out-of-order read-read accesses to the same word like in Section 4.6.4: when a read executes and finds that a later read to the same address has already been sent to the bus (because it found the *Unknown* state), the hardware forces a second bus access.

4.7.2 V-State Transitions for a Word

Figure 4.7 shows how the V-state of a word changes. We break the transitions into two figures. Figure 4.7(a) shows the transitions of the word as its line moves in and out of the cache; Figure 4.7(b) shows the transitions while the line is in the cache.

Starting with Figure 4.7(a), as the line is brought-in on a read miss, the hardware operates on the Vulcan metadata of the referenced word, recording any unsafe dependence. Hence, the word is loaded as *Recorded*. The other words in the line (i.e., "not-referenced" words) are loaded as either *Recorded* — if their address cannot be in any of the other processor's SCVQs — or as *Unknown* otherwise. This functionality is easily supported by adding one control wire in the bus for each word in a line. During the bus transaction, the processors also check the addresses of all the not-referenced words in the line against their bloom filter. If any processor finds a match for a given

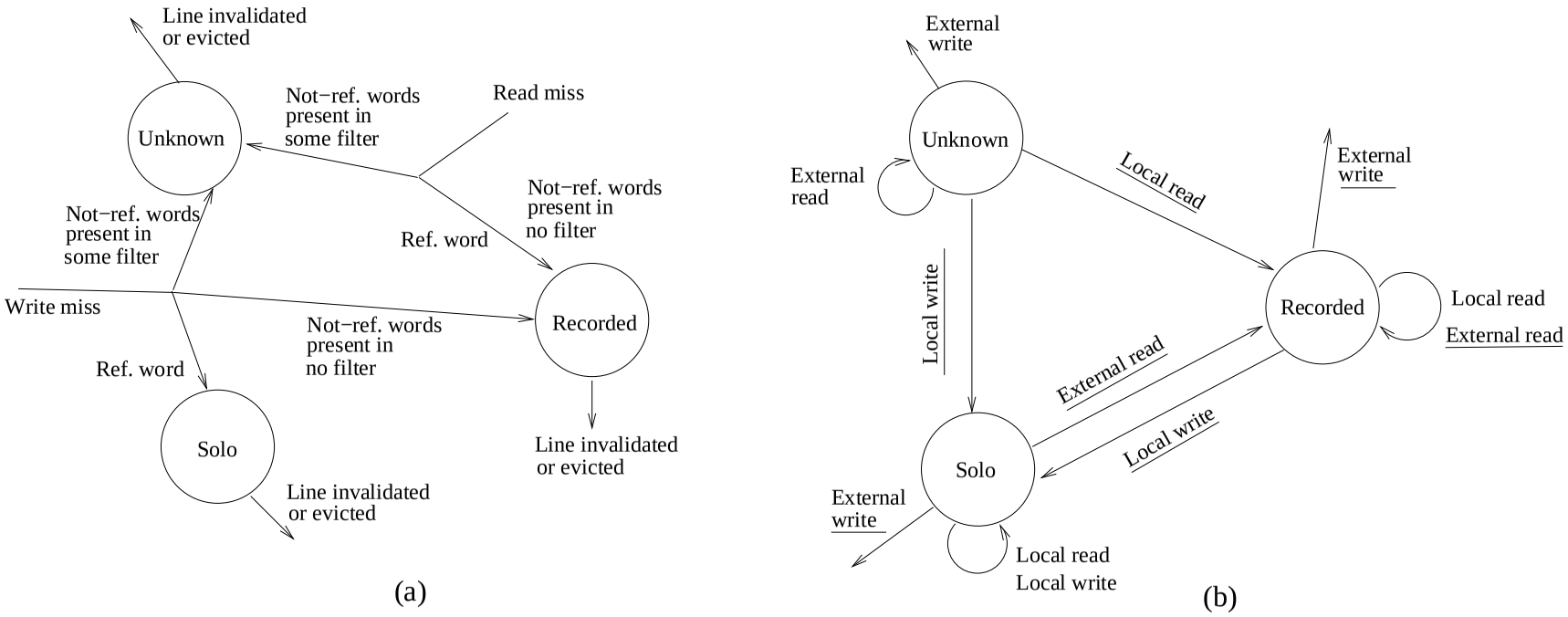


Figure 4.7: V-state transitions for a word. In chart (a), *ref* stands for referenced. In chart (b), the underlined transitions involve SCVQ monitoring, metadata exchange and update and, hence, need a bus access (possibly piggybacked on a coherence action).

word, it sets the control wire for that word. If the wire for a particular word is not set by the end of the bus transaction, it means that no processor can possibly have the word in its SCVQ, and the word's V-state in the requester is set to *Recorded*.

Hardware-prefetched lines work seamlessly. The algorithm for the not-referenced words is applied to all the words in the line.

If the line is brought-in on a write miss, the state becomes *Solo* for the referenced word. For the other words, the bloom filters are checked as above and the state is set as *Recorded* if there is no match in any filter or *Unknown* if there is. Note that the V-state information is lost when the line leaves the cache. This occurs when the line is evicted or invalidated by an external write to any one of its words.

In Figure 4.7(b), the line is in the cache in any valid state. The transitions that require metadata update in potentially both dependence source and destination and, therefore, need a bus access are underlined. Such access is possibly piggybacked on a coherence action in the bus. Consider a *Solo* word. The local processor can read and write it without metadata access. An external read requires metadata update and the transition to *Recorded* V-state. Consider a *Recorded* word. A local read is silent. A local write and an external read cause a bus access with metadata update, bringing the local state to *Solo* or *Recorded*, respectively. Finally, in an *Unknown* word, a local read and write cause a bus access and bring the word to *Recorded* and *Solo*, respectively. An external read is ignored. In all states, an external write was described above and invalidates the line.

4.7.3 SCVQ Implementation

The SC Violation Queue (SCVQ) is a FIFO queue that contains the Vulcan metadata for local unsafe loads and stores. As a load or store enters the pipeline, an SCVQ entry is allocated, holding the next Serial Number (*SN*) and default values for *AS* and *AD* (∞ and the preceding access' *AD*, respectively). The *AS* and *AD* are updated later, when (1) the load or store executes, and (2) an external access creates a dependence with the load or store. The latter event involves a bus access and a search of the SCVQ.

The SCQV implementation is shown in Figure 4.8. The metadata is stored in a FIFO circular queue. Since the bus can trigger an access to any address, the bus accesses the queue through a hash table on word addresses. With this design, it is easy for the processor to allocate and deallocate entries in FIFO order. It is also easy for the bus to find the entries that match a certain address. Finally, a write bus transaction that invalidates an SCVQ entry simply marks the entry as "empty" without moving it.

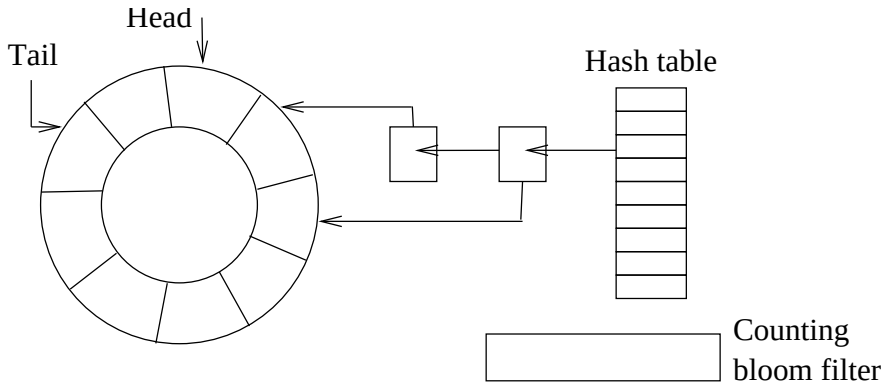


Figure 4.8: Implementation of the SC Violation Queue (SCVQ).

On a bus transaction, we need to look-up the SCVQ for an address match. To reduce the number of useless look-ups, we cannot rely on filtering by the cache snooper. This is because an SCVQ match may occur even if the corresponding line is no longer in the cache. Consequently, Vulcan hashes all the word addresses currently in the SCVQ in a counting bloom filter [7]. This structure uses counters to allow the removal of an individual hashed-address. As entries are inserted and removed from the SCVQ, the addresses are added and removed from the filter. Then, bus transactions check the filter for a match before initiating an SCVQ access.

Inserting and removing addresses from the filter is not in a critical path. Insertion can occur any time from when the address of the reference is known until when the load/store completes. In the meantime, the metadata is not up-to-date anyway. Removal can be done lazily, since false positive matches in the filter are harmless.

4.8 Discussion and Limitations

Vulcan is attractive because it is a very precise detector of SC violations — unlike some past work that focuses on the related (but different) problem of detecting data races [24, 38, 43]. In addition, Vulcan needs no compiler support. Finally, it provides precise exceptions on SC violation detection, since it detects the exact reference that closes the dependence cycle.

The current Vulcan implementation has some limitations that can be fixed with additional features. The first one is that it uses words as the finest grain of access. If the program has byte-level accesses, Vulcan may suffer from false positives and false negatives. To solve this problem, Vulcan needs to keep information (including V-states) and operate at byte granularity. A second limitation is that the current design focuses on dependence cycles involving only two processors. In practice, this is not a major limitation because three-processor dependence cycles are extremely rare — they require the overlapping of three races. In fact, all of the related existing work focuses on two-processor interactions (e.g., [8, 10, 14, 31]). Anyway, the Vulcan framework can be mapped to several-processor cycles by propagating the *AS/AD* along the dependence arrows, as opposed to just sending *SN*.

Other coherence protocols and cache hierarchy organizations may require changes to the design presented. Vulcan correctly supports load forwarding in the processor. However, the current design does not support misspeculated loads — i.e., loads executed past mispredicted branches. In the current design, these loads could trigger a dependence cycle. One approach to support such loads is for Vulcan to be re-designed to wait until the loads become non-speculative before performing metadata updates.

Finally, Vulcan's hardware does not scale well to large numbers of processors because it has to be designed to check for all-to-all processor interactions. However, this is not a significant limitation. It is well known that small numbers of processors are perfectly adequate to test for concurrency bugs [34].

4.9 Conclusion

SC violations arising from overlapping data races intertwined in a cycle are virtually always harmful bugs: they result from totally-unintuitive memory access orders and cannot be reproduced using a single-stepping debugger. While past work has focused on looking for data races as their proxies, a new taxonomy of data races that we introduced shows that more precise detection methods are needed.

To address this problem, this paper proposed Vulcan, the first hardware scheme that dynamically detects SC violations in a program running on a relaxed-consistency machine *precisely*. Vulcan's idea is to leverage cache coherence protocol transactions to dynamically detect cycles in memory access orderings across threads. When one such cycle is about to occur, an exception is triggered, providing the exact architectural state. Vulcan suffers from no false positives or false negatives, induces negligible execution overhead, requires no help from the compiler, and only takes as input the program executable. Experimental results showed that Vulcan detects *three new bugs* in very popular codes. Specifically, it found SC violations in the Pthread and Crypt libraries, and in the fmm SPLASH-2 program. We have reported the bugs to the developers. Also, we showed that Vulcan had a negligible execution overhead, which makes it suitable for on-the-fly use in production runs.

Chapter 5

Evaluation

In this chapter we evaluate our work. The high level goal of this evaluation is to show that - (i) the algorithm works, (ii) the technique detects both known and possibly new, unreported bugs, (iii) they require moderate hardware and, (iv) they have negligible execution time overhead. The chapter will start with the evaluation of SigRace, then AtomTracker and finally it will end with Vulcan.

5.1 SigRace

To evaluate SigRace, we consider four issues: (1) the signature configuration, which determines the number of false positives, (2) the block size and number of entries in each BlockHistoryQueue[i], which determine the window of monitored execution, (3) the effectiveness of SigRace in detecting data races, and (4) the overheads of SigRace. In the following, we first overview the experimental setup and then consider each issue in turn.

5.1.1 Experimental Setup

Since we are interested in the high-level parameters of SigRace, we use the PIN [40] binary instrumentation tool to design a simulator of the SigRace hardware, and run the applications on a real 8-processor shared-memory machine. This approach has the benefit of execution-driven simulation without incurring the slow speeds of typical cycle-accurate simulators. Table 5.1 shows the default parameters used in the simulation.

We model an 8-core chip multiprocessor where 32-Kbyte L1 caches are connected in a mul-

Number of processors: 8 L1 cache size: 32 Kbytes L1 cache line size: 64 bytes Coherence protocol: MESI Checkpoint interval: 1 M committed instr./proc.	Timestamp size: 8 x 20 = 160 bits Signature size: 2 Kbits each R and W Block size: 2,000 committed instr. BlockHistoryQueue[i] size: 16 entries
Benchmarks: SPLASH2 kernels: FFT, Cholesky, LU SPLASH2 applications: Barnes, Volrend, Ocean, Radiosity, Raytrace, Water-ns, Water-spatial PARSEC kernels: Dedup, Streamcluster PARSEC applications: Blackscholes, Fluidanimate, Swaptions	

Table 5.1: Default parameters used in the evaluation.

tistage network and kept coherent with a MESI cache coherence protocol. The timestamp size is very conservatively set to 160 bits. The default values for the size of signatures, block, and BlockHistoryQueue[i] are set according to the sensitivity analyses presented later. We take periodic global checkpoints. A checkpoint is created as soon as a processor has committed 1 M instructions. We use the checkpointed information as a starting point of our Re-execution and Analysis algorithms.

We evaluate SigRace with the SPLASH2 and PARSEC [4] benchmarks. These benchmarks are representative of parallel workloads and exhibit a wide variety of memory access patterns. For SPLASH2, we use the default inputs, while for PARSEC, we use the *simmedium* input size. We report data for 10 SPLASH2 and 5 PARSEC benchmarks. As shown in Table 5.1, we separate them into SPLASH2 kernels, SPLASH2 applications, PARSEC kernels, and PARSEC applications.

5.1.2 Signature Configuration

We test multiple signature configurations, denoted as $B_i\_S_j$. We first partition the address into 2 portions. The possible configurations are the B_i in Table 5.2. Then, we use multiple Bloom filters in parallel using the *H3* hash function as in [64] — half of them process one portion while the other half the other. The configurations are the S_i in Table 5.3.

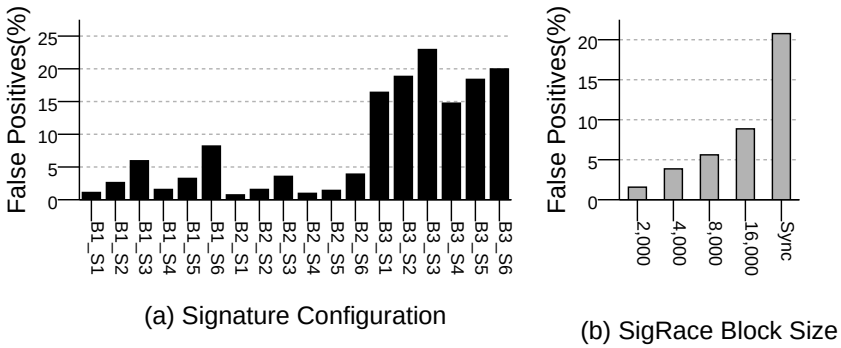
Configuration	Address Partition	
	LSB	USB
B_1	8	24
B_2	10	22
B_3	16	16

Table 5.2: Address partitions. LSB and USB stand for Lower and Upper Sliced Bits.

Configuration	# of Bloom Filters (k)	Bits per Bloom Filter (n)	Sig Size ($k \times n$)
S_1	16	256	4Kbit
S_2	16	128	2Kbit
S_3	16	64	1Kbit
S_4	8	512	4Kbit
S_5	8	256	2Kbit
S_6	8	128	1Kbit

Table 5.3: Signature organizations.

We run the applications and count the number of signature intersections that indicate a collision while there is none. The ratio of this number over the total number of signature intersections is the false-positive rate. Figure 5.1(a) shows the average false-positive rate of the applications for our default parameters. In the rest of the work, we use $B_2\_S_2$, where the false-positive rate is 1.57%.



(a) Signature Configuration

(b) SigRace Block Size

Figure 5.1: False positive rate versus signature configuration (a) and versus block size (b).

5.1.3 Block and BlockHistoryQueue[i] Size

If we choose a large SigRace block then, with the same BlockHistoryQueue[i] (BHQ[i]) size, we can monitor a larger instruction window for possible data races. However, as the block size increases, the signature false-positive rate also increases. Figure 5.1(b) shows the false-positive rate for different block sizes beyond our default of 2,000 committed instructions. *Sync* means terminating a block only at synchronizations. We see that larger blocks induce more false positives.

For a given block size, if we increase the number of entries in BHQ[i], we cover a larger instruction window. However, we have to do more signature operations and the BHQ takes more area.

To evaluate these issues, we run the applications with different numbers of entries in BHQ[i] and different block sizes. When the RDM checks an incoming signature against a BHQ[i], the hardware operates on each of the entries in the BHQ[i] until it finds a block that is a predecessor of the incoming one. If there is such a predecessor, then SigRace does not lose any race detection opportunity. We call this event a *Hit*. Otherwise, SigRace loses race detection opportunity beyond the oldest entry in BHQ[i]. We are interested in the execution window that starts at the previous checkpoint and ends at the block just before the oldest entry in BHQ[i]. We call it the *Lost Detection Window*.

Figure 5.2(a) shows the lost detection window as a percentage of the checkpoint interval, while Figure 5.2(b) shows the hit rate of a signature against a BHQ[i], and Figure 5.2(c) shows the number of timestamp comparisons in a BHQ[i] per signature until hitting in the BHQ[i] or exhausting all full BHQ[i] entries. All figures have the same X axis and share the same legend.

We see that, as the number of BHQ[i] entries increases, the lost detection window decreases (Figure 5.2(a)) and the hit rate increases (Figure 5.2(b)). However, we have to do more timestamp comparisons until a hit or BHQ[i] exhaustion (Figure 5.2(c)), and the BHQ takes more area. On the other hand, for a fixed number of BHQ[i] entries, as the block size increases, we lose less window (Figure 5.2(a)), the hit rate increases (Figure 5.2(b)) and the number of comparisons decreases

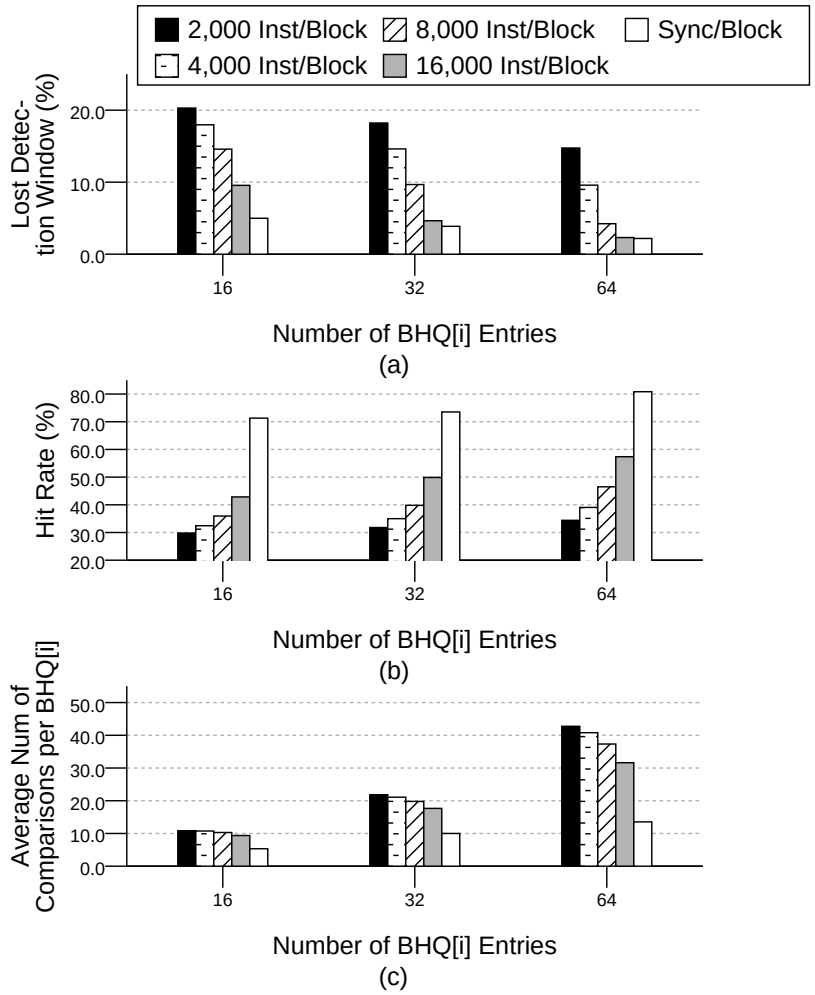


Figure 5.2: Lost detection window (a), hit rate (b), and number of timestamp comparisons (c) for different numbers of BHQ[i] entries and block size. All figures share the same legend.

(Figure 5.2(c)) — however, we saw in Figure 5.1(b) that false positives increase. Overall, we choose as default a block size of 2,000 committed instructions and 16 entries in BHQ[i]. This leads to an average of 20% loss in detection window.

5.1.4 SigRace Effectiveness

Data Race Detection

To assess SigRace's effectiveness, we use it to find (i) existing data races in our applications and (ii) races that we inject in the applications. We also simulate a cache-based race detector, namely a version of ReEnact [58] with per-word timestamps (*W-ReEnact*). Table 5.4 shows the results.

Columns 2-7 (*Finding Existing Races*) list the number of races found by *Ideal Sigrace*, SigRace, and W-ReEnact. Ideal SigRace is a SigRace where each BHQ[i] keeps information for *all* the blocks between consecutive checkpoints — rather than for 16 blocks as in SigRace. Races are identified by the two instructions involved in the race and the address accessed. The table counts both static and dynamic races. Dynamic races are the dynamic instances of static races.

The table shows that 8 of the applications have data races. These races include, for example, reads of shared structures outside a critical section before accessing them inside the critical section. They are likely to be all benign races. However, we believe that it is important for any race detector to detect even benign races. This is because, often, benign races are a symptom that the code has a bug or something that the programmer does not understand. In any case, as described in Section 2.5.2, if the programmer wants SigRace to skip checking for these races, he can mark the code with *collect_off*.

The table shows that SigRace detects 90 static and 47,000 dynamic races. Compared to W-ReEnact, SigRace detects on average 29% more static races and 107% more dynamic races. SigRace's substantially higher effectiveness is due to its ability to monitor a longer window of program at a time. Finally, compared to Ideal SigRace, SigRace detects on average 95% of the static races and 26% of the dynamic ones.

We also inject races. For each application, we perform 25 runs. In each run, we randomly eliminate one dynamic lock-unlock pair or one dynamic barrier. Since the Swaptions code synchronizes with fork/joins, we could not subject it to this experiment. While these are contrived examples, they provide some insight.

Application	Finding Existing Races						Finding Injected Races				
	Ideal SigRace		SigRace		W-ReEnact		Racy	Static Races Found		Runs w/ Races Found	
	Stat	Dyn	Stat	Dyn	Stat	Dyn	Runs	SigRace	W-ReEnact	SigRace	W-ReEnact
FFT	–	–	–	–	–	–	25/25	600	150	25	25
Cholesky	16	19964	16	3539	16	388	3/25	2	2	1	1
LU	–	–	–	–	–	–	25/25	28	75	25	25
Barnes	11	4416	11	719	6	419	1/25	3	1	1	1
Volrend	27	26846	27	11607	18	6858	23/25	345	74	23	21
Ocean	1	29	1	29	1	6	7/25	8	8	7	7
Radiosity	15	59307	15	16951	12	14660	8/25	29	11	8	6
Raytrace	4	30	4	17	3	12	21/25	66	53	21	21
Water-ns	–	–	–	–	–	–	5/25	2	4	1	2
Water-spatial	8	82	4	27	2	3	3/25	6	6	3	3
Dedup	–	–	–	–	–	–	3/25	0	0	0	0
Streamcluster	13	68566	12	14307	12	436	6/25	7	2	5	2
Blackscholes	–	–	–	–	–	–	0/25	0	0	0	0
Fluidanimate	–	–	–	–	–	–	12/25	95	90	12	12
Swaptions	–	–	–	–	–	–	–	–	–	–	–
Total	95	179240	90	47196	70	22782	142/350	1191	476	132	126

Table 5.4: Effectiveness of SigRace and ReEnact with per-word timestamps in finding existing races and injected races.

Columns 8-12 (*Finding Injected Races*) show the detection capability of SigRace and W-ReEnact. Column 8 (*Racy Runs*) shows the fraction of those 25 runs that actually created races. Then, Columns 9-10 show the number of static races found by SigRace and W-ReEnact, respectively. We see that, on average, SigRace finds 150% more static races than W-ReEnact. This again shows the higher effectiveness of SigRace. Interestingly, there are two applications where W-ReEnact finds more races (LU and Water-ns). This is because, while SigRace typically monitors a longer program window, there are cases when lines remain in the caches for a long time. In this case, W-ReEnact can detect racing accesses that are far apart in the code (over 50,000 instructions apart in these examples). In general, it can be argued that races where the accesses are far apart are least dangerous, since the chances that these accesses appear in reverse order in a different run are lower. Finally, Columns 11-12 show the number of runs in which SigRace and W-ReEnact found at least one race. Again, the number for SigRace is higher.

Opportunity to Detect Data Races

SigRace has an advantage when addresses are in BHQ[.] and not in caches, while W-ReEnact has an edge in the opposite case. In this section, we estimate the frequency of each case. For simplicity, in this experiment only, signatures encode line addresses.

Of all the cache lines with shared data being displaced or invalidated from a cache, Figure 5.3(a) shows the fraction whose address is strictly present (not just due to aliasing) in the corresponding BHQ[i]. The figure shows the average for different cache sizes and application sets. For the 32KB default cache, the weighted average fraction is ≈59%. Then, Figure 5.3(b) shows the number of displacements or invalidations of lines with shared data per million instructions executed. For the 32KB default cache, the weighted average can be shown to be ≈2,800. Overall, roughly speaking, compared to SigRace, W-ReEnact loses detection opportunity for 0.59×2,800=1,652 lines per million instructions.

Given a block being displaced from a BHQ[i], Figure 5.3(c) shows the fraction of addresses in the block's signatures that are not anywhere else in BHQ[i] and that are in the cache. For the

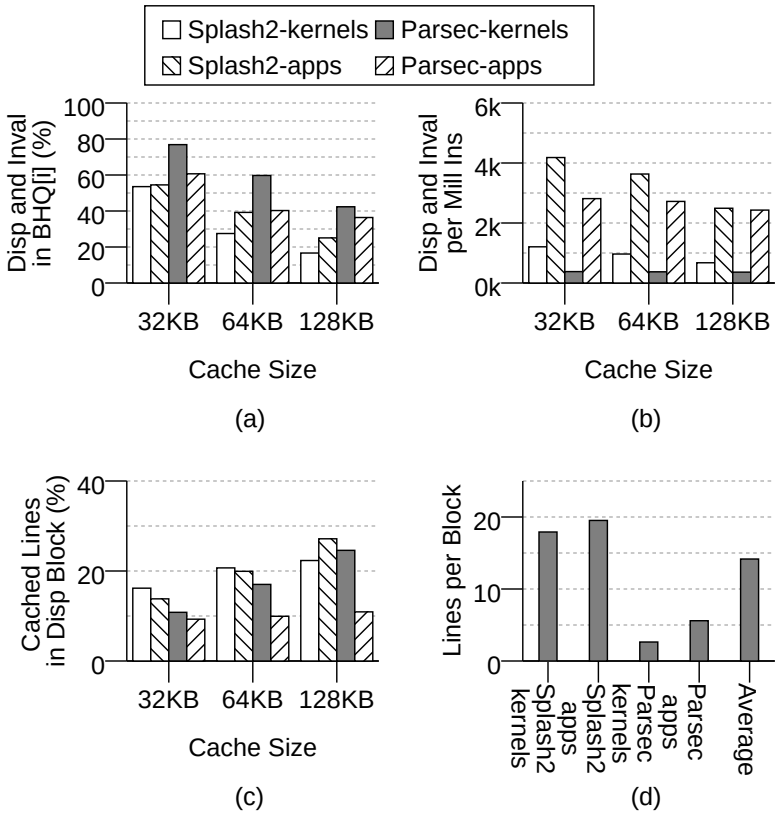


Figure 5.3: Opportunities for SigRace and W-ReEnact to detect races. Charts (a), (b), and (c) share the same legend.

32KB cache, the weighted average fraction is ≈13%. Figure 5.3(d) shows the number of addresses of lines with shared data that are encoded in the signatures of one block. This number is on average 14. Overall, since SigRace executes ≈500 blocks per million instructions, compared to W-ReEnact, SigRace loses detection opportunity for $0.13 \times 14 \times 500$=910 lines per million instructions. While these numbers give approximate information only, they show W-ReEnact loses more opportunities.

5.1.5 SigRace Overheads

We estimate the instruction, SRAM memory, bandwidth, and checkpointing overheads of SigRace. To estimate the instruction overhead, we run each application until the first true data race is fully

analyzed. In the process, some false positives may occur. We count as instruction overhead all the instructions executed in Re-execution and Analysis modes to characterize the true data race and all the false positives found from the beginning of the program until that point. We stop after analyzing the first true race because then the programmer would stop execution. If the application has no true data race, we insert one in a random location.

Figure 5.4(a) shows the resulting instruction overhead as a percentage of committed instructions. The average bar is the mean of all the applications. The overhead depends on several things, most notably how far from the previous checkpoint is the conflict detected, and the rate of false positives. We see that, on average, the instruction overhead is 22%. The large majority of it is due to re-execution. About two thirds of it is caused by false positives.

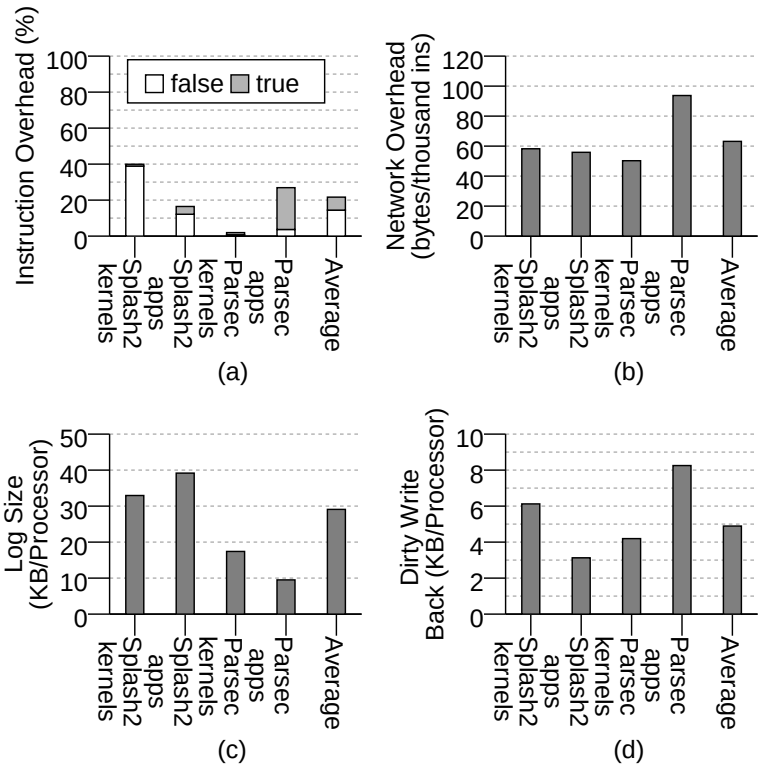


Figure 5.4: Instruction (a), bandwidth (b), and checkpoint-related (c and d) overheads.

From Figure 2.5, we see that the main SRAM memory overhead of SigRace per processor includes: a 16-entry BHQ[i] in the RDM (each entry containing a timestamp and a R and W

signature), one extra timestamp and R and W signatures, the TRT, and the Conflict signature. Since timestamps are 160 bits and signatures 2K bits, this results in 8512 bytes in the RDM and 808 bytes in the cache hierarchy — independently of the cache size.

To compute the bandwidth overhead of SigRace, we count how many bytes of timestamp-signature messages (compressed) are deposited on the network. Figure 5.4(b) shows such number per 1,000 instructions committed. We see that, on average, the bandwidth overhead is 63 bytes per thousand committed instructions.

Finally, we measure some overheads of checkpointing every 1M instructions. As per Section 2.4.3, the memory controller saves the value overwritten by every first memory update. Figure 5.4(c) shows that, on average, this amounts to 29KB of log per processor between checkpoints. Also, at the point of checkpoint, the dirty lines in the cache are written back. As shown in Figure 5.4(d), this corresponds to, on average, 4.8KB of writebacks per processor.

5.2 AtomTracker

In this section, we evaluate AtomTracker-I and AtomTracker-D. We implement the AtomTracker-I algorithm in C++, and run it on traces of parallel applications generated by a Pin [40] tool. After AtomTracker-I determines the ARs, we use AtomTracker-D to find violations. We evaluate two implementions of AtomTracker-D, namely a software one using Pin and a hardware one using a whole-system simulator of the multicore architecture of Section 3.6 based on Simics [42]. The parameters of the multicore architecture simulated are shown in Table 5.12.

We use three representative applications (Apache, MySql, and Mozilla) and focus on eight documented atomicity violation bugs. These bugs have been used in the evaluation of past works like AVIO [35], MUVI [32], and PSet [75]. They are described in Table 5.6. Of these bugs, the three Mozilla bugs, MySql#2, and MySql#3 are multi-variable atomicity bugs; the rest are single-variable atomicity bugs. We also use six SPLASH-2 codes to characterize AtomTracker: three kernels (FFT, LU-con, and LU-non-con) and three applications (Barnes, FMM and Water-ns).

Multicore	4 cores at 4 GHz
Core	4 issue out-of-order
L1 cache (private)	32 KB, 4 way, 2 cycle lat.
L2 cache (private)	512 KB, 8 way, 12 cycle lat.
Cache line	64B
Memory	80 cycle round trip lat.
Network	Bus
Bus bandwidth	128B/cycle
Coherence protocol	MESI
Signatures	2Kbit each like in [50]

Table 5.5: Multicore architecture evaluated.

Lastly, we also use three synthetic microbenchmarks to evaluate AtomTracker.

Our evaluation aims to (i) demonstrate the training methodology, (ii) show the bug detection ability, (iii) characterize the false positives, (iv) quantify the execution overhead, and (v) show the completeness of our design.

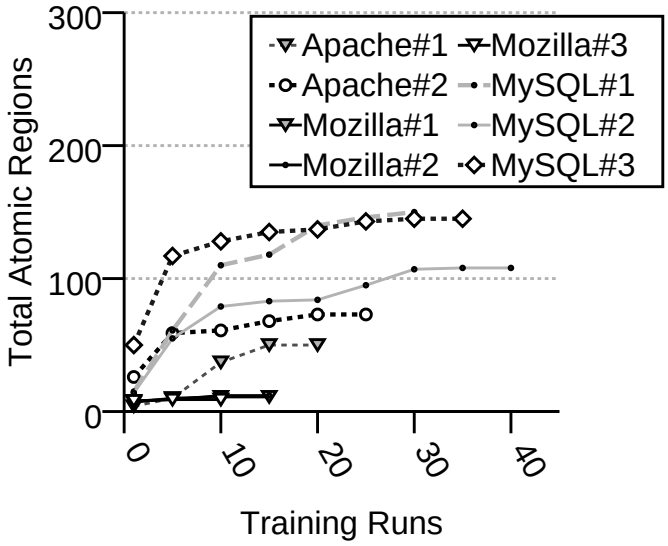
5.2.1 Training Sensitivity

To make AtomTracker effective, we need enough training runs to obtain a good set of ARs. To obtain many training runs, we change the program inputs and also get different interleavings for the same input. For MySql, we change the number of concurrent requests to the server. For Apache#2, we use *httperf* to send different numbers of concurrent requests, while for Apache#1, we use different numbers of calls to *wget* to fetch different numbers of web pages concurrently. For Mozilla, we wrote a driver that calls the buggy library with different parameters and different numbers of iterations. In all tests, we check that we do not trigger the bugs. This is easily done because all the bugs manifest themselves with either a program failure or a wrong output. Fortunately, bugs are hard to exercise because they require special interleavings. For all the applications, we stop the training as soon as we get 5 consecutive training runs that generate no new ARs.

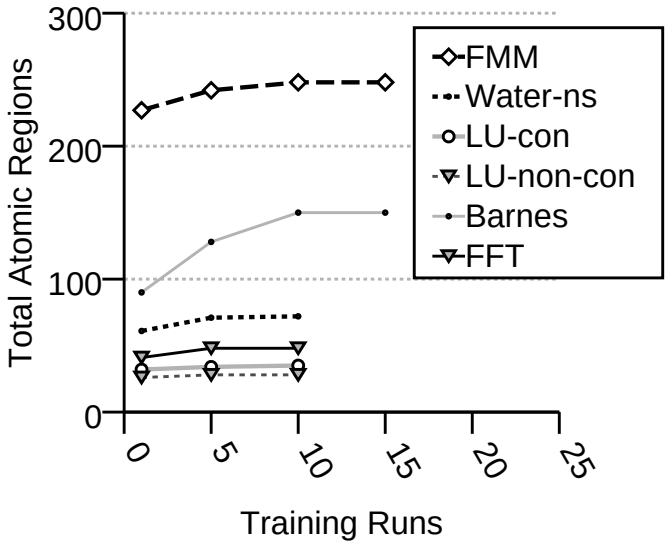
Figure 5.5 shows the convergence of ARs in the commercial (a) and the SPLASH-2 (b) codes as we execute training runs. We perform from 10 to 40 training runs. The SPLASH-2 codes converge faster than the commercial codes because of their smaller size. Table 5.7 shows the average size

Bug	Version	Files Involved	# Variables	Description
Apache#1	2.0.48	mod_log_config.c	Single	Unprotected read and write of buffer length can corrupt log file.
Apache#2	2.0.46	mod_mem_cache.c	Single	Unprotected read and write of reference counter can cause null pointer dereference.
Mozilla#1	0.8	jsstr.h, jsstr.c	Multiple	Non-atomic update of total number of strings and total string length permit them to be inconsistent.
Mozilla#2	0.8	jsinterp.h, jsinterp.c	Multiple	Non-atomic access of cache structure can cause cache and empty flag to be inconsistent.
Mozilla#3	0.9	jsdhash.c	Multiple	Concurrent access of *entryCount* and *removedCount* can cause the table to incorrectly shrink.
MySql#1	4.0.12	log.cc, sql_insert.cc	Single	Unprotected close and open of database bin log can cause some actions not to be logged.
MySql#2	3.23.56	sql_insert.cc, sql_delete.cc	Multiple	Non-atomic update of rows and bin log can cause wrong order of logging. Shown in Figure 1.2
MySql#3	4.0.16	slave.cc	Multiple	Non-atomic read of *log_file_name* and *log_file* can cause slave sql thread to fail.

Table 5.6: Bug descriptions.



(a)



(b)

Figure 5.5: Convergence of ARs for the commercial (a) and the SPLASH-2 (b) codes as we execute training runs.

of the resulting ARs, in number of source code lines that access shared variables (the lines that the programmer will check) and in number of shared variables. On average, ARs in the commercial codes have 52 lines and access 114 shared variables, whereas ARs in the SPLASH-2 codes have 5 lines and access 122 variables.

App	# Lines of Code	# Shared Variables
Apache#1	44.3	62.4
Apache#2	76.4	125.6
Mozilla#1	62.2	82.9
Mozilla#2	83.8	197.1
Mozilla#3	46.4	102.1
MySql#1	43.9	230.4
MySql#2	44.3	82.4
MySql#3	17.0	31.8
Avg	52.3	114.3
SPLASH-2 kernels	4.6	215.5
SPLASH-2 apps	4.7	28.2
Avg	4.7	121.8

Table 5.7: Average AR size.

5.2.2 Bug Detection Ability

After AtomTracker-I infers the set of ARs, we provide the bug-triggering input and run AtomTracker-D. Our AtomTracker-D algorithm detects the resulting atomicity violation in every case. Table 5.8 compares the effectiveness of AtomTracker to that of AVIO [35], MUVI [32], and PSet [75] — based on data on the same bugs reported in the papers of those schemes.

AVIO and PSet are reported to catch the three single-variable atomicity bugs. However, since, by construction, they cannot catch multi-variable bugs, they cannot catch the three *Mozilla* bugs, *MySql#2*, or *MySql#3*. MUVI focuses on catching multi-variable data races. Consequently, it does not handle the bugs with a single variable, namely *Apache #1*, *Apache #2*, and *MySql #1*. The other five bugs are both multi-variable data races and multi-variable atomicity violations. The MUVI paper reports that MUVI catches the *Mozilla* bugs and *MySql#3*, but not *MySql#2*. The reason

Bug	Atomicity Violation Detected?			
	AtomTracker	AVIO	MUVI	PSet
Apache#1	Yes	Yes	No	Yes
Apache#2	Yes	Yes	No	Yes
Mozilla#1	Yes	No	Yes	No
Mozilla#2	Yes	No	Yes	No
Mozilla#3	Yes	No	Yes	No
MySql#1	Yes	Yes	No	Yes
MySql#2	**Yes**	**No**	**No**	**No**
MySql#3	Yes	No	Yes	No

Table 5.8: Comparison of bug detection ability. Recall that the *Mozilla* bugs, *MySql#2*, and *MySql#3* are multi-variable atomicity violations.

why *MySql #2* (shown in Figure 1.2) is undetected by MUVI but is detected by AtomTracker is as follows. MUVI fails because the correlation between *t->rows* and *binlog* is conditional: *t->rows* and *binlog* do not always need to be accessed together; only when *t->rows* is modified at the end, *binlog* needs to be modified atomically with it. This atomicity relation is easily extracted by AtomTracker-I by examining execution traces. MUVI does not extract this relation.

5.2.3 False Positives

AtomTracker is a heuristic-based approach and, therefore, subject to False Positives (FPs). To evaluate this issue, we apply AtomTracker-D to five bug-free runs per program — obtained by changing the inputs. Table 5.9 shows the average number of FPs observed per run, for three different scenarios: (i) software implementation, (ii) hardware implementation where, rather than signatures, we use an unbounded buffer to store addresses in the AVM, and (iii) hardware implementation with signatures as in Section 3.6.

The software implementation of AtomTracker-D has an average of only 0.8 and 1.6 FPs per run for the commercial and SPLASH-2 codes, respectively. These few FPs are due to undertraining. The more we train, the better the AR accuracy will be, and hence the fewer the FPs will be. The hardware implementation has two additional sources of FPs: false sharing due to using cache line addresses and aliasing due to signatures. Column 3 of Table 5.9 includes only the impact of false

App	FP in SW impl	FP in HW impl	
		Unbounded buffer	Signatures
Apache#1	1.8	2.0	2.0
Apache#2	2.6	10.4	14.4
Mozilla#1	0.4	1.8	1.8
Mozilla#2	0.0	3.0	6.0
Mozilla#3	0.0	0.0	0.0
MySql#1	0.6	12.2	15.0
MySql#2	0.8	7.6	9.6
MySql#3	0.2	17.2	18.0
Avg	0.8	6.8	8.4
SPL2 kernels	0.0	14.7	15.3
SPL2 apps	3.3	12.7	17.6
Avg	1.6	13.7	16.4

Table 5.9: False positives in different scenarios.

sharing, since signatures are replaced by an unbounded address buffer. This leads to an average of 6.8 and 13.7 FPs per run for the commercial and SPLASH-2 codes, respectively. When we use signatures and, therefore, include the aliasing effect as well, the average FPs per run is 8.4 and 16.4.

Overall, the number of FPs in AtomTracker-D is comparable to other schemes. For example, for similar commercial codes, AVIO has 7 FPs per code (compared to 8.4 in AtomTracker-D), and SVD has 3.2 FPs per M instructions (compared to 0.16 FPs per M instructions in AtomTracker-D).

5.2.4 Execution Time Overhead

Table 5.15 shows the overhead of the complete hardware and software implementations of AtomTracker-D. The overhead of the hardware one is measured in increase in execution time and in network traffic. The execution time increases by an average of only 0.2% and 4.0% for the commercial and SPLASH-2 codes, respectively. This makes AtomTracker-D suitable for *production runs*. This low overhead results from running the algorithm in hardware in the AVM. The main source of overhead is the additional network traffic caused by first-time accesses in an AR that are intercepted by

the cache. This causes on average 3.3% and 9.2% more traffic for the commercial and SPLASH-2 codes, respectively. The SPLASH-2 codes induce higher traffic because they have relatively more data sharing.

App	HW impl		SW impl	
	Execution time increase (%)	Traffic increase in bytes (%)	Slowdown (x)	
			Due to Pin	Total
Apache#1	0.1	1.3	8.7	80.4
Apache#2	0.1	1.0	6.9	74.1
Mozilla#1	0.1	5.5	2.9	14.6
Mozilla#2	0.5	6.3	1.3	2.1
Mozilla#3	0.1	2.7	1.5	1.9
MySql#1	0.1	4.2	6.2	15.9
MySql#2	0.1	1.5	7.5	13.9
MySql#3	0.3	3.8	1.8	2.8
Avg	0.2	3.3	4.6	25.7
SPL2 kernels	2.2	9.0	106.4	573.1
SPL2 apps	5.8	9.4	19.3	256.1
Avg	4.0	9.2	62.9	414.6

Table 5.10: Execution overhead of AtomTracker-D.

The software implementation slows down, on average, 26x and 415x the commercial and SPLASH-2 codes, respectively. Of this, Pin accounts for a 5x and 63x slowdown, respectively. Since our software implementation is highly unoptimized, these slowdowns should only be considered an upper bound. They can easily be reduced with a better implementation. Still, they are acceptable for in-house testing, especially those for the commercial codes.

5.2.5 Components of AtomTracker-I

As described in Section 3.4.2, AtomTracker-I uses a preprocessing pass that collects Critical Section (CS) and Loop (LP) information. To evaluate the contribution of this pass in finding ARs, we use three microbenchmarks for which we know the actual ARs. We cannot use our main applications because we do not know the correct ARs there. The three microbenchmarks, shown in Table 5.11, implement a linked list, a producer-consumer pattern, and an FFT. They have 17, 4,

and 14 ARs, respectively. Table 5.11 shows the fraction of the correct ARs that are inferred by our algorithms. We consider four cases: the complete AtomTracker-I (ATI), ATI without the critical section information (ATI-CS), ATI without the loop information (ATI-LP), and ATI without either (ATI-CS-LP). From the average numbers, we see that AtomTracker-I identifies all the ARs. Without CS or LP information, AtomTracker-I identifies only 79% or 90% of them. So, both types of information are needed.

Micro-benchmark (# of ARs)	ARs Inferred by AtomTracker-I (ATI) versions			
	ATI (% of correct)	ATI - CS (% of correct)	ATI - LP (% of correct)	ATI - CS - LP (% of correct)
LinkedList (17)	100.0	58.8	94.1	52.9
ProdCons (4)	100.0	100.0	75.0	75.0
FFT (14)	100.0	78.6	100.0	78.6
Avg (11.7)	100.0	79.1	89.7	68.8

Table 5.11: Impact of the AtomTracker-I preprocessing pass.

5.3 Vulcan

In this section, we evaluate Vulcan. Our goal is to (1) validate its effectiveness in detecting SC violations, (2) determine the size of the SCVQ, and (3) assess Vulcan's overhead in terms of network traffic and execution time.

5.3.1 Experimental Setup

We model Vulcan's architecture using a cycle-accurate execution-driven simulator. The simulator is based on SESC [62], and models the processor and memory systems of a multicore with four superscalar processor cores. The processors are out-of-order, three-retire wide, and execute under release consistency. They have a simple cache hierarchy composed of a private L1 cache and a shared L2 cache. Table 5.12 shows the baseline architecture parameters. We conservatively use 4 bytes for Serial Numbers.

Architecture	4 processors in a multicore chip
Proc. freq.	2.0GHz
Proc. pipeline	out-of-order, 3-retire wide
ROB size	32 entries
L1 cache	32KB WB, 4-way asso., 2-cycle RT, 32B line
L2 cache	1MB WB, 8-way asso., 11-cycle RT, 32B line
Coherence	Snoopy-based MSI protocol
Bus OC	Coherence access: 2-cyc request, 4-cyc reply Metadata access: 2-cyc request, 2-cyc reply
Main memory	500-cycle RT
Vulcan param.	SCVQ size: 256 entries, serial number: 4 B

Table 5.12: Baseline multicore architecture evaluated. RT and OC stand for round trip and occupancy, respectively.

We used three sets of applications for the evaluation. The first set includes implementations of some concurrent data structures and mutual exclusion algorithms that have SC violations. They are taken from [8, 10]. The second set includes some reported SC violation bugs and bug patterns from the open source libraries. The last set includes 8 applications from the SPLASH-2 suite. The first two sets have known SC violations and are used to evaluate Vulcan's effectiveness. The last set has long-executing applications, supposedly free of SC violations, and is used to estimate Vulcan's overheads. Table 5.13 describes the applications.

Set	Program	Description
Conc. Algo.	Dekker	Algorithm for mutual exclusion
	Lazylist	List-based concurrent set
	Snark	Nonblocking double-ended queue
	Harris	Nonblocking set
Bug Kernels	Pthread_cancel from glibc	Unwind code after canceling thread needs memory barrier [3]
	Crypt_util from glibc	Small table initialization code needs memory barrier [1]
	DCL bug	Kernel using double checked locking without fences
Full Apps.	SPLASH-2	8 programs form the SPLASH-2 application suite

Table 5.13: Applications analyzed.

5.3.2 Detection Ability

To test Vulcan's SC violation detection ability, we run each application multiple times — 100 times for the concurrent algorithms and bug kernels, and 5 times for the SPLASH-2 codes. In each run, we generate different interleavings by forcing the processors to miss some random number of fetch cycles. We identify each observed SC violation by the pair of instructions that create the dependence that closes the cycle. Such a pair may create multiple dynamic instances of the same SC violation during an execution. For each application, we compute, over all the runs, the number of *unique* and *total* number of SC violations observed. This information is shown in Table 5.14, for cache lines of 1 and 8 words (i.e., 32 bytes). Note that the table only includes the fmm application from SPLASH-2, since Vulcan found no SC violation in the other SPLASH-2 codes.

Program	Line Size (Words)	SC Violations Found		Total Runs
		Unique	Total	
Dekker	1	1	1982	100
	8	1	224	100
Lazylist	1	0	0	100
	8	1	150	100
Snark	1	1	745	100
	8	1	1467	100
Harris	1	0	0	100
	8	1	18	100
Pthread_cancel	1	2	298	100
	8	2	142	100
Crypt_util	1	2	564	100
	8	2	130	100
DCL	1	2	648	100
	8	1	2	100
fmm	1	1	2	5
	8	3	18	5

Table 5.14: SC violations found in various applications.

Columns 3 and 4 of Table 5.14 show the number of unique and total SC violations found with two different cache line sizes over all the runs. Vulcan detects SC violations in all of the concurrent algorithms and bug kernels. It also finds one in the fmm code from SPLASH-2. Of the unique

SC violations in the table, *three are new, unreported SC violation bugs*. They are one in each of Pthread _cancel, Crypt_util, and fmm. We discuss them in Section 5.3.3.

The number of unique SC violations observed is 1-3 per application. However, the total number of them can soar to many hundreds for some codes. In addition, the number of unique and, especially, total SC violations depends substantially on the line size. This shows that this bug is highly dependent on the actual timing of the events in the processor. Overall, we see that Vulcan is very effective. Note that, with more runs, new interleavings may occur and Vulcan may find more SC violations.

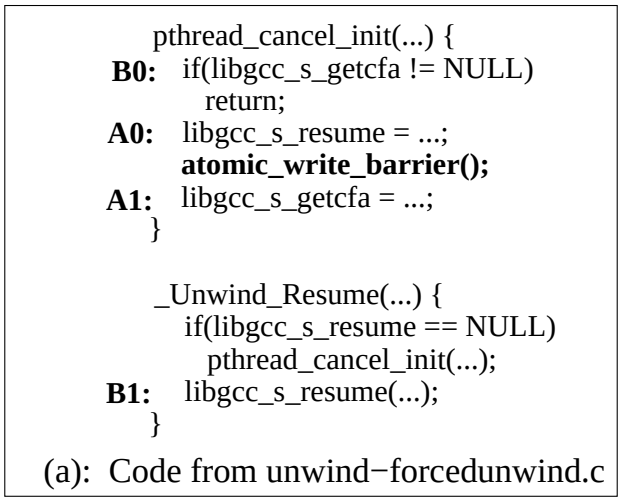
5.3.3 Three New SC Violations Found

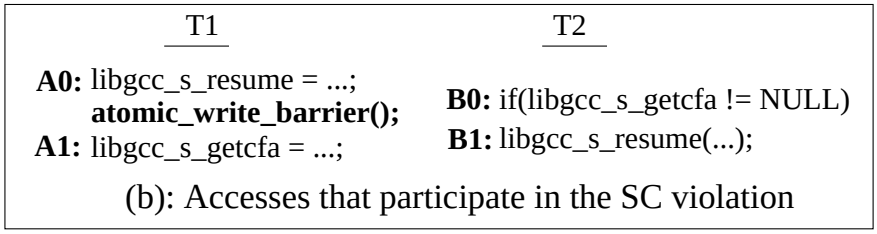
New SC Violation in the pthread Library

One of the SC violations in the *pthread_cancel* kernel of Table 5.14 is Bug ID 2644 in the Redhat bug database, which had been fixed by the developers. After running Vulcan, we found a *new SC violation in the bug fix*. We reported the new bug and its fix to the developers, who have most recently implemented it.

Figure 5.6 shows the bug. Figure 5.6(a) shows the *pthread_cancel_init* and *_Unwind_Resume* functions. Assume that thread T1 is inside *pthread_cancel_init*, and about to initialize function pointers *libgcc_s_resume* (in A0) and *libgcc_s_gtecfa* (in A1). Before it does so, another thread (T2) is in *_Unwind_Resume* and calls *pthread_cancel_init*. There, it finds *libgcc_s_gtecfa* already non-null (in B0), returns from *pthread_cancel_init* and uses *libgcc_s_- resume* (in B1). Due to an SC violation, *libgcc_s_resume* is still uninitialized and the program crashes.

The references involved are shown in Figure 5.6(b), together with the fence (write barrier) that the developers inserted in an attempt to fix the bug. This code is the same as Figure 4.1 except for the fence. Unfortunately, the fence only prevents the reordering of A0 and A1. In an RC or PowerPC memory model, B0 and B1 can get reordered as in Figure 5.6(c). The condition in B0 is true but B1 finds the function uninitialized. This is the SC violation that crashes the code. To fix





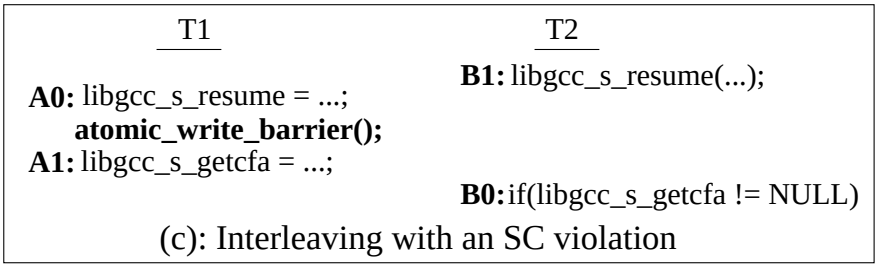


Figure 5.6: New SC violation found in the glibc pthread library.

this, we also add a fence between B0 and B1.

New SC Violation in the crypt Library

A similar case occurs for *crypt_util*. One of its SC violations in Table 5.14 is Bug ID 11449 in the database, which had also been incorrectly fixed by the developers. After running Vulcan, we found a new SC violation in the bug fix. We reported the new bug and its fix to the developers. They declined to fix it because it only happens in memory models more relaxed than Intel's x86.

Figure 5.7 shows the bug. Figure 5.7(a) shows the code of function *__init_des_r*, which uses DCL to initialize shared tables. Assume that thread T1 enters the function, grabs the lock and is about to initialize the table *eperm32tab* (in A0) and then set *small_tables- _initialized* (in A1). Another thread (T2) enters the function, finds *small_tables_initialized* set (in B0) and proceeds to use *eperm32tab* (in B1). Unfortunately, due to an SC violation, *eperm32tab* is still uninitialized and the program behaves incorrectly.

The references involved are shown in Figure 5.7(b), together with the fence that the developers added to fix the bug. This example is like the one in Section 5.3.3: another fence between B0 and B1 is needed to avoid SC violations.

New SC Violation in fmm from SPLASH-2

Vulcan finds three new SC violations in fmm, caused by a single flag dependence racing against three pairs of references. The code for one of the racing pairs is shown in Figure 5.8. Inside the *Set-Colleagues* function, a thread (T2) sets structure *colleagues* (in A0) and then flag *construct_synch* (in A1); another one spins on the flag (in B0) and then uses the structure (in B1). This is like the pattern in Figure 4.1 and an SC violation occurs. In the fmm code, the flag was declared as volatile. However, in C, while volatile prevents compiler optimizations, it does not prevent reordering by the hardware.

This SC violation affects the precision of the program's output because thread T1 uses "old data". However, since fmm is an N-body problem, the output might still be acceptable. Still,

```
      _init_des_r(...){
B0:     if(small_tables_initialized==0){
          lock;
          if(small_tables_initialized)
            goto Done;
A0:       eperm32tab[...]= ...;
          atomic_write_barrier();
A1:       small_tables_initialized=1;
  Done: unlock;
        }
        ...
B1:     ... =eperm32tab[...];
    }
```

(a): Code from crypt_util.c

T1	T2
A0: eperm32tab[...]= ...;	
atomic_write_barrier();	**B0:** if(small_tables_initialized==0){
A1: small_tables_initialized=1;	**B1:** ... =eperm32tab[...];

(b): Accesses that participate in the SC violation

Figure 5.7: New SC violation found in the glibc crypt library.

T1	T2
SetColleagues(...) {	SetColleagues(...) {
B0: while(b−>construct_synch==0);	**A0:** b−>colleagues[...] =...;
B1: ... = parent_b−>colleagues[...];	**A1:** child_b−>construct_synch=1;
}	}

Code from construct_grid.c

Figure 5.8: New SC violation found in fmm from SPLASH-2.

we argue that this is a serious bug because the programmer can hardly reason about the code and estimate how much the bug affects the output in all possible cases. This bug can be fixed by either placing a fence between the two references in each thread, or by using a synchronization instruction to access the flag.

5.3.4 Size of the SC Violation Queue (SCVQ)

To size the SCVQ, we need to know the number of unsafe accesses that individual processors maintain over time. Consequently, we count the average and maximum number of unsafe accesses that a processor keeps over the course of each application. We use only SPLASH-2 applications because the others are too small to provide any useful information. For our measurements, we take a sample every time a memory operation is issued. Also, to gain more insight, we additionally collect measurements on the average and maximum number of pending accesses. These are loads and stores that are already issued but not yet performed, as defined in Section 4.6.2. Recall from that section that an access remains unsafe at least while pending and often beyond that. Figure 5.9 shows the results for each application and for the average of them.

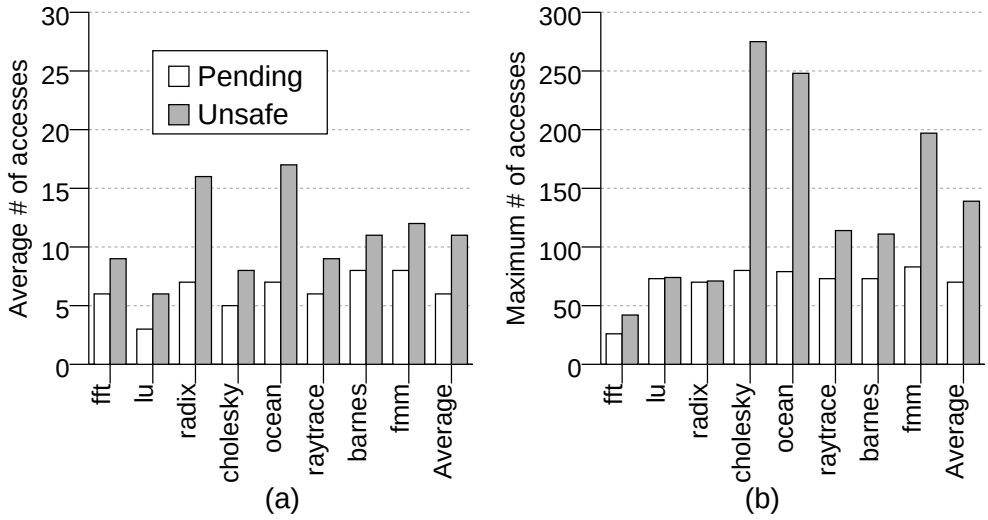


Figure 5.9: Number of pending and unsafe accesses: average values (a) and maximum values (b).

Figure 5.9(a) shows that the average number of unsafe accesses ranges from 6 to 17. This is a small number, and is about double of the average number of pending accesses. Since accesses are typically bursty, the maximum number of unsafe accesses is higher. Across applications, it ranges from 50 to 270 (Figure 5.9(b)). If we average out all the applications, this number is also about double of the maximum number of pending accesses.

Overall, to be very conservative, we size the SCVQ with 256 entries. Most of the time, only about 10 or so entries are in use. Sometimes, many more entries exist. If the processor were to need more entries than 256, then it temporarily stalls. This only happens 170 times in cholesky, which issues about 147 million memory accesses. Therefore, such stall is negligible.

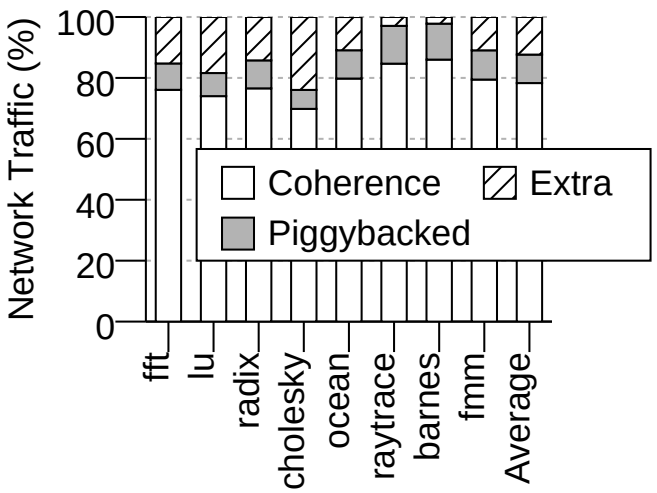


Figure 5.10: Breakdown of total bus traffic in bytes.

5.3.5 Network Traffic & Execution Overhead

In Vulcan, the operations on the metadata and the accesses to the SCVQ are not in the critical path of execution. Therefore, Vulcan's overhead comes from the additional bus traffic that it induces. This traffic comes from two sources: (i) the information that it piggybacks on many of the normal coherence transactions on the bus and (ii) the Metadata bus accesses that it induces (Section 4.6.4).

In both transactions, Vulcan sends a Serial Number in the request (conservatively, 4 bytes), and both a Serial Number and a Performed Point in the response (again conservatively, a total of 8 bytes).

To see the magnitude of this traffic, Figure 5.10 breaks down the total bytes of traffic in the bus for each application. The categories are: traffic induced by a Vulcan-free execution (*Coherence*), traffic piggybacked by Vulcan on the normal coherence (*Piggybacked*), and traffic in Metadata bus accesses (*Extra*). We can see that the effect of Vulcan is modest: on average, *Piggybacked* accounts for 9.4% of the traffic and *Extra* for 12.4%.

Appl.	Total Ins (M)	Mem Ops (%)	Total Bus Acc (M)	Metad. Bus Acc (%)	Exec. Time Over (%)
fft	6.4	25.9	0.4	32.4	4.9
lu	145.2	20.7	1.2	34.4	3.8
radix	45.0	16.1	2.0	32.5	0.7
cholesky	469.0	31.8	34.5	38.9	7.0
ocean	143.2	28.6	21.5	26.7	5.8
raytrace	277.1	34.3	3.1	8.8	4.2
barnes	2785.6	43.2	30.7	6.9	2.7
fmm	3713.1	39.2	19.2	25.8	2.6
Avg.	948.1	30.0	14.1	25.8	4.0

Table 5.15: Overhead of Vulcan.

To understand the impact of this traffic on the execution time, recall from Table 5.12 that the bus occupancy of a Vulcan-free transaction is 2 and 4 cycles for request and reply, respectively. Given the implied bus bandwidth, we assume that the additional bytes piggybacked by Vulcan do not increase these occupancies. However, the bus is also used by Metadata accesses, whose bus occupancy is set to 2 cycles for both requests and replies. The contention induced by these accesses causes Vulcan's execution overhead.

Table 5.15 shows the resulting overhead. For each application, it shows the total number of instructions executed, the fraction of them that are memory operations, and the total number of bus accesses in the application. Column 5 shows what fraction of all the bus accesses are Metadata bus

accesses. We see that such fraction is, on average, about 26%. Therefore, about one quarter of the accesses are Metadata accesses. Finally, the last column shows the execution time overhead caused by Vulcan. Such overhead is consistently small for all the applications. Its value ranges from 1% to 7%, with an average of 4%. Overall, therefore, we conclude that the execution overhead of Vulcan is negligible enough to allow its on-the-fly use in production runs.

Chapter 6

Conclusion

In order to make parallel programming easier, we need good debugging support This thesis strives to achieve that goal. It proposes to add some hardware that can be used both during development time and during production run to detect various types of concurrency bugs. First, it proposes to detect data races, which is the most common form of concurrency bugs. Then it goes on to detect atomicity violation bugs which is harder and more difficult to debug than data races. Finally, it tries to detect a special type of bugs that can violate sequential consistency which is arguably the hardest type of concurrency bugs.

References

[1] Sources bugzilla. Bug 11449. `http://sources.redhat.com/bugzilla/show_bug.cgi?id=11449`.

[2] Sources bugzilla. Bug 133773. `https://bugzilla.mozilla.org/show_bug.cgi?id=133773`.

[3] Sources bugzilla. Bug 2644. `http://sources.redhat.com/bugzilla/show_bug.cgi?id=2644`.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7):422–426, 1970.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Ann. Euro. Symp. on Algo.*, Sep 2006.

[8] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Prog. Lang. Des. and Impl.*, Jun 2007.

[9] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *Comp. Aid. Veri.*, Jul 2008.

[10] Jabob Burnim, Koushik Sen, and Christos Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *Tools and Algo. for the Const. and Ana. of Sys.*, July 2011.

[11] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *International Symposium on Computer Architecture*, June 2007.

[12] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[13] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[14] Kaiyu Chen, Sharad Malik, and Priyadarsan Patra. Runtime validation of memory ordering using constraint graph checking. In *High Perf. Comp. Arch.*, Feb 2008.

[15] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming Language Design and Implementation*, June 2002.

[16] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[17] Andrew Deorio, Ilya Wagner, and Valeria Bertacco. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *High Perf. Comp. Arch.*, Feb 2009.

[18] Yuelu Duan, Xiaobing Feng, Lei Wang, Chao Zhang, and Pen-Chung Yew. Detecting and eliminating potential violations of sequential consistency for concurrent C/C++ programs. In *Code Gen. and Opt.*, Mar 2009.

[19] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Op. Sys. Des. and Impl.*, Feb 2010.

[20] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Int. Conf. on SuperComp.*, Jun 2003.

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

[22] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multi-threaded programs. In *Symposium on Principles of Programming Languages*, January 2004.

[23] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation*, June 2003.

[24] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *Symp. on Para. Algo. and Arch.*, Jul 1991.

[25] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *International Conference on Software Engineering*, May 2008.

[26] Intel Corporation. Intel Thread Checker. http://www.intel.com, 2008.

[27] Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space programs. *Jour. Paral. Dist. Comp.*, Nov 1996.

[28] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Tran. on Comp.*, July 1979.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, 1978.

[30] Jaejin Lee and David A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.*, Aug 2001.

[31] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient sequential consistency using conditional fences. In *Par. Arch. and Compil. Tech.*, Sep 2010.

[32] 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 *Symposium on Operating Systems Principles*, October 2007.

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[34] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Arch. Supp. for Prog. Lang. and Op. Sys.*, Mar 2008.

[35] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[36] Brandon Lucia, Luis Ceze, and Karin Strauss. Finding concurrency bugs with context-aware communication graphs. In *International Symposium on Computer Architecture*, December 2009.

[37] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *International Symposium on Computer Architecture*, June 2010.

[38] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Int. Symp. on Comp. Arch.*, Jun 2010.

[39] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *International Symposium on Computer Architecture*, June 2008.

[40] Chi-Keung Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, June 2005.

[41] Ewing Lusk, James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ross Overbeek, James Patterson, and Rick Stevens. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.

[42] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[43] Daniel Marino, Abhayendra Singh, Todd Millstein, Madanlal Musuvathi, and Satish Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *Prog. Lang. Des. and Impl.*, Jun 2010.

[44] Albert Meixner and Daniel J. Sorin. Dynamic verification of sequential consistency. In *Int. Symp. on Comp. Arch.*, Jun 2005.

[45] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[46] Chi Cao Minh et al. An effective hybrid transactional memory system with strong isolation guarantees. In *International Symposium on Computer Architecture*, June 2007.

[47] Tipp Moseley, Dirk Grunwald, Daniel A. Connors, Ram Ramanujam, Vasanth Tovinkere, and Ramesh Peri. LoopProf: Dynamic techniques for loop detection and profiling. In *Workshop on Binary Instrumentation and Applications*, October 2006.

[48] Abdullah Muzahid et al. SigRace: signature-based data race detection. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 337–348, New York, NY, USA, 2009. ACM.

[49] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 287–297, 2010.

[50] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: Signature-based data race detection. In *International Symposium on Computer Architecture*, June 2009.

[51] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, June 2007.

[52] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Prog. Lang. Des. and Impl.*, Jun 2007.

[53] Robert H. B. Netzer and Barton P. Miller. Detecting data races in parallel program executions. In *In Workshop on Advances in Languages and Compilers for Parallel Computing*, pages 109–129, 1990.

[54] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Principles and Practice of Parallel Programming*, April 1991.

[55] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, June 2003.

[56] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *International Symposium on Foundations of Software Engineering*, November 2008.

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

[58] Milos Prvulovic and Josep Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *International Symposium on Computer Architecture*, June 2003.

[59] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, May 2002.

[60] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[61] Sriram Rajamani, G. Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani. ISOLATOR: Dynamically ensuring isolation in comcurrent programs. In *Arch. Supp. for Prog. Lang. and Op. Sys.*, Mar 2009.

[62] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

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

[64] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proc. of the 40th Annual Int'l Symp. on Microarchitecture*, pages 123–133, 2007.

[65] 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(4):391–411, 1997.

[66] Douglas C. Schmidt and Tim Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Patt. Lang. of Prog. Des. Conf.*, 1996.

[67] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Prog. Lang. and Sys.*, April 1988.

[68] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, June 2004.

[69] Sun Microsystems. Sun Studio Thread Analyzer. http://developers.sun.com/sunstudio, 2007.

[70] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent Java programs. In *Prin. and Pract. of Para. Prog.*, Jun 2005.

[71] Christoph von Praun and Thomas R. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2001.

[72] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 2006.

[73] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Conference on Programming Language Design and Implementation*, June 2005.

[74] Luke Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *International Symposium on High Performance Computer Architecture*, February 2007.

[75] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *International Symposium on Computer Architecture*, June 2009.

[76] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Int. Symp. on Comp. Arch.*, Jun 2009.

[77] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Symposium on Operating Systems Principles*, October 2005.

[78] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-assisted lockset-based race detection. In *International Symposium on High Performance Computer Architecture*, February 2007.

Appendix

Here we outline the proof of why Vulcan detects all SCVs between any two processors.

Theorem 1: An access C_i of processor P_C is Safe when ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$), for all processors $K \neq C$.

Proof: C_i can participate in an SCV with either an earlier access C_{i-l} (Case 1) or a later access C_{i+m} (Case 2) for $l, m > 0$. Here, an earlier or later access is defined in terms of program order. The two cases are shown in Figure 1.

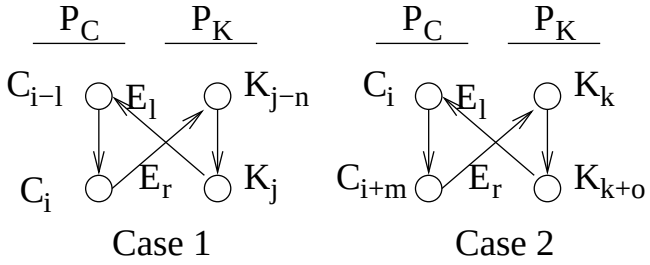


Figure 1: Possible cases for SCV.

Case 1: If edge E_r occurs first, C_i is not safe until it provides data to K_{j-n}. Here, data can be either value stored in that address or invalidation ack. C_i can provide data to other accesses until the program ends but this pattern can happen only until all previous accesses C_{i-l} where $1 \leq l \leq i$ are performed. So, C_i is not safe until all previous accesses are performed i.e. until $PP[P_C] \geq SN_{C_{i-1}}$. However, if edge E_l occurs first, C_i is not safe until C_{i-l} is performed and all K_{j-n} for $0 \leq n \leq j$ is performed. This pattern can happen with all previous accesses C_{i-l} where $1 \leq l \leq i$. That means C_i is not safe until all previous accesses are performed (just like the previous scenario) and all accesses of P_K upto $MAX(SN_{K_j})$ where $SN_{K_j} = src(C_{i-l})$ for $1 \leq l \leq i$ is performed i.e $MAX(SN_{K_j}) \leq PP[P_K]$. We should note that $AD_{C_i}[P_K] = MAX(SN_{K_j})$.

Therefore, the condition for safety in Case 1 is $AD_{C_i}[P_K] \leq PP[P_K]$ and $PP[P_C] \geq SN_{C_{i-1}}$.

Case 2: If edge E_l occurs first, C_i is not safe until it is performed. At that time, all the necessary updates will be done so that when E_r occurs the violation gets detected. Now, if edge E_r occurs first C_i is not safe until it is performed at which point the violation will be detected.

Now, Case 1 requires all previous accesses of C_i to be performed and $AD_{C_i}[P_K]$ to be less than or equal to $PP[P_K]$. Case 2 requires C_i to be performed. So, combining them, we can say that C_i is safe when all accesses upto C_i are performed and $AD_{C_i}[P_K] \leq PP[P_K]$. If we generalize this to all the processors of the system, an access C_i is safe when ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$), for all processors $K \neq C$.

Theorem 2: In order to form an SCV cycle with two dependences, their source references have to be unsafe with respect to the destination processors.

Proof: This can be proved by contradiction. Let us assume that one of the dependences has a source that is safe (w.r.t. the destination processor) at the time of the dependence and it forms an SCV cycle with another dependence whose source is unsafe (w.r.t the destination processor). According to the definition of a safe access, once an access becomes safe (w.r.t. a processor), no future dependence from this access to any access of that processor can cause an SCV. But this contradicts our previous assumption. Therefore, this proves the theorem.

Theorem 3: Given all the unsafe dependences, Vulcan detects all the SCVs between the processors.

Proof: Referring to Figure 1, an access C_i can have SCV with any of its previous accesses (Case 1) or any of its later accesses (Case 2). Without loss of generality, we can assume that dependence edge to/from C_i completes the cycle. For Case 1, C_i needs to store *src* information of all of its previous accesses. This is done by taking the MAX of SN_{K_j} where $K_j = src(C_{i-l})$ for $1 \leq l \leq i$. This is precisely what AD_{C_i} stores. Therefore, SCV will detected as a violation of condition for AD_{C_i} (i.e. a destination has to be larger than this). For Case 2, C_i needs to store *dest* information of all of its later accesses. This is done by taking the MIN of SN_{K_k} where $K_k = dest(C_{i+m})$ for $1 \leq m \leq (N_C - i)$ where N_C is the total number of accesses by processor P_C. This is precisely

what AS_{C_i} stores. Therefore, SCV will detected as a violation of condition for AS_{C_i} (i.e. a src has to be smaller than this). Thus, an SCV will be detected in both cases. This means that if we have all the dependences' information, Vulcan can detect all SCVs between the processors. Now, Theorem 2 proves that SCV occurs only among unsafe dependences. This, along with the fact that we keep the metadata of every unsafe access ensures that Vulcan detects all the SCVs between the processors