Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically [*]

Abdullah Muzahid[†], Shanxiang Qi, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Past work has focused on detecting data races as proxies for Sequential Consistency (SC) violations. However, most data races do not violate SC. In addition, lock-free data structures and synchronization libraries sometimes explicitly employ data races but rely on SC semantics for correctness. Consequently, to uncover SC violations, we need to develop a more precise technique.

This paper presents *Vulcan*, the first hardware scheme to precisely detect SC violations at runtime, in programs running on a relaxed-consistency machine. The scheme leverages cache coherence protocol transactions to dynamically detect cycles in memory-access orders across threads. When one such cycle is about to occur, an exception is triggered. For the conditions considered in this paper and with enough hardware, Vulcan suffers neither false positives nor false negatives. In addition, Vulcan induces negligible execution overhead, requires no help from the software, and only takes as input the program executable. Experimental results show that Vulcan detects *three new SC violation bugs* in the Pthread and Crypt libraries, and in the fmm code from SPLASH-2. Moreover, Vulcan's negligible execution overhead makes it suitable for on-the-fly use.

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 execute in some global sequence as if the threads where multiplexed on a uniprocessor [18]. In practice, however, current hardware overlaps, pipelines, and reorders the memory accesses of threads. As a result, a program's execution can be unintuitive.

As an example, consider Figure 1(a). Processor P_A allocates a variable and then sets a flag; later, P_B tests the flag and, if set, it uses the variable. While the particular interleaving in Figure 1(a) produces expected results, the interleaving in Figure 1(b) does not. In here, the hardware reorders the *completion* of the stores in the two statements in P_A. In this unlucky interleaving, P_B ends up using an unallocated variable. This order is an SC Violation (SCV).

From the hardware point of view, several conditions must be met for an SCV to occur. First, we need to have at least two data races — i.e., races on variables *buff* and *init* in the example. Secondly, these races must be of a very special type: they must be *overlapping* in time and *intertwined* in a manner that can form a cycle [30]. For two threads, it requires a pattern like that in Figure 2(a) where, if we follow program order, the two threads reference the same two

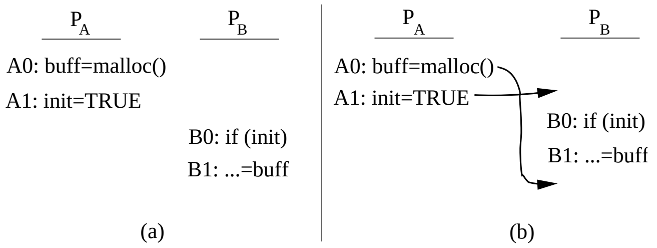


Figure 1. Example of an SC violation.

variables in opposite orders, and each variable is written at least once. Finally, the order of the references in these two racing pairs *has to form a cycle* at runtime. This is shown in Figure 2(b), where we have arbitrarily picked reads and writes: *A1* must occur before *B0* and *B1* must occur before *A0*. This is exactly what happened in Figure 1(b), where *y* was *init* and *x* was *buff*.

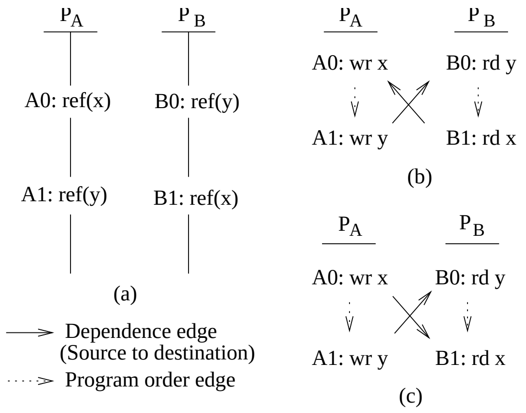


Figure 2. SC violations are possible.

Note, however, that if the timing at runtime is such that at least one of the two dependence arrows occurs in the opposite direction, there is no SCV. For example, Figure 2(c) shows the case when *A1* executed before *B0* but *A0* executed before *B1*. Since there is no cycle, SC is not violated. This case corresponds to the timing in Figure 1(a).

Data race patterns that cause SCVs are sometimes found in double-checked locking constructs [29], some synchronization libraries, and code for lock-free data structures.

Detecting SCVs is important because, in practically all cases, they are harmful, clear-cut bugs. The reason is that, as the example in Figure 1(b) shows, they require memory access orders that contradict a programmer's intuition. In addition, the programmer cannot reproduce them using a single-stepping debugger.

Past work has attempted to find SCVs by focusing on detecting data races (e.g., [4, 9, 15, 16, 23, 24, 32]). However, as we just saw, using data races as proxies for SCVs is very imprecise. The specific race pattern and interleaving required for an SCV is not necessarily

[*] This work was supported in part by the National Science Foundation under grants CNS 0720593, CCF 1012759 and CNS 1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).

[†] Abdullah Muzahid is now with the University of Texas at San Antonio. His e-mail address is muzahid@cs.utsa.edu.

common. In large commercial codes, conventional race-detection tools typically flag many data races, often causing the programmer to spend time examining races that are much less likely to cause code malfunctioning than SCVs [13, 26].

A second reason for not using data races as proxies is that we may want to find SCVs in codes that have intentional data races, such as in lock-free data structures. We may want to debug the code for SCVs, while being less concerned about non-SC-violating races. Here, a race-detection tool would not be a good instrument to use. If we want to detect SCVs, we need to precisely zero-in on the types of data races and interleavings that cause them.

Given the importance of these bugs and the difficulty in isolating them, this paper contributes with *Vulcan*, the first hardware scheme to precisely detect SCVs at runtime, in programs running on a relaxed-consistency machine. Vulcan leverages cache coherence protocol transactions to dynamically detect cycles in memory access orders across threads. When a cycle is about to occur, an exception is triggered, providing information to debug the SCV.

The Vulcan design in this paper focuses on finding cycles of overlapping races between only two processors — since cycles involving three and more processors are much rarer. In addition, it does not consider speculative loads from mispredicted branch paths. Moreover, it is not concerned with SCVs due to compiler transformations — Vulcan only reports SCVs due to hardware-initiated reference reordering. Within these constraints, and with large-enough hardware structures, Vulcan suffers neither false positives nor false negatives.

Vulcan's approach has several advantages: it induces negligible execution overhead, requires no help from the software, and only takes as input the program executable. Experimental results show that Vulcan detects *three new bugs* in popular codes. Specifically, it finds SCVs 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.

We also contribute with a new taxonomy of data races.

This paper is organized as follows: Section 2 gives a background; Section 3 introduces a taxonomy of data races; Sections 4 and 5 present Vulcan; Section 6 outlines its limitations; Section 7 evaluates Vulcan; and Section 8 discusses related work.

2. 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 interleaving. It is virtually always a harmful bug, since it is the outcome of an unintuitive execution. Moreover, it is difficult to debug because single-stepping debuggers cannot reproduce it.

Shasha and Snir [30] show what causes an SCV: overlapping data races where 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 2(a) showed the required program pattern for two threads (where each variable is written at least once) and Figure 2(b) showed the required order of the dependences at runtime (where we assigned reads and writes to the references arbitrarily).

If at least one of the dependences occurs in the opposite direction (e.g., Figure 2(c)), no SCV occurs. In addition, if the code of the two threads references the two variables in the same order (Figure 3(a)), no SCV is possible — no matter how the hardware reorders these references at runtime. For example, in Figure 3(b), no SCV can occur, no matter the direction of the inter-thread dependences.

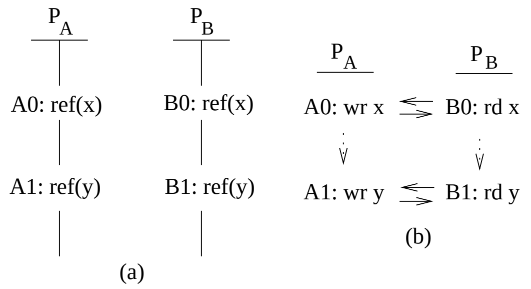


Figure 3. SC violations are not possible.

Given the pattern in Figure 2(a), Shasha and Snir [30] prevent the SCV by placing one fence between references *A0* and *A1*, and another between *B0* and *B1*. Their algorithm to find where to put the fences is called the Delay Set.

The commonly used Double-Checked Locking (DCL) [29] 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 in Figure 1(a). Because the code is typically involved, programmers often miss putting the two fences needed.

Data races and SCVs are very different, and programs have more data races than SCVs. However, past work has focused on detecting data races as proxies for SCVs. Specifically, one line of work detects incoming coherence messages on data that has local outstanding loads or stores. This work started with Gharachorloo and Gibbons [15] and now includes many aggressive speculative designs (e.g., [4, 9, 16, 32]). Another line of work detects a conflict between two concurrent synchronization-free regions. This includes DRFx [24] and Conflict Exceptions [23]. In general, all of these works look for a data race with two accesses that occur within a short time — but still, only a *single race*. Overall, while focusing on these races may be a good way to discard many irrelevant ones, 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 [12, 14, 17, 19, 31]. Since the compiler has limited information, these approaches tend to be very conservative and result in substantial slowdowns. Lin *et al.* [20] hide much of the resulting fence delay with architectural support.

Lin *et al.* [21] have recently proposed a design to support SC in a relaxed-consistency machine. While its goal is different than Vulcan's, it also involves the analysis of race cycles. We discuss it in Section 8. Finally, in the program testing and verification domains, there are proposals to detect SCVs by checking the semantic correctness of programs, or by collecting traces and then, off-line, applying reordering rules [6, 7, 8]. While such techniques are promising, they are typically limited to small-sized codes and are performed statically or as an off-line pass. Vulcan's goal is to detect SCVs in large codes on-the-fly and with negligible overhead. More details on related work are presented in Section 8.

3. A Taxonomy of Data Races

To assess the relationship between data races and SCVs, we develop a taxonomy of data races. We examined the bug databases of popular programs such as Apache, MySQL, and Mozilla, and collected all the data-race bugs we could find. Since these are races reported by users, we know that they caused the program to malfunction. Table 1 lists the applications and the number of reported data races.

Application	# Reported Data Races	# Multi-Races	# SCV Races	# DCL SCVs
Apache	24	5	5	5
MySQL	13	1	1	1
Mozilla	11	2	1	1
Redhat (glibc)	2	2	2	1
Java SDK	2	1	1	1
PostgreSQL	1	0	0	0
Pbzip2	1 from [33]	0	0	0
Windows kernel	1 from [13]	0	0	0
Isolator bench.	1 from Isolator [27]	0	0	0
Total	56	11	10	9

Table 1. Reported data races that we studied.

Overall, we found 56 reported race-based bugs. 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, SCV races are classified into those that are DCLs [29] and those that are not.

Table 1 shows the breakdown of the bugs per application. We see that, of the total 56 reported race bugs, 11 are multi-races (20%). Of these, 10 can cause SCVs (91%). The only one that, due to its reference pattern cannot ever create an SCV is in Mozilla [2]. Of the 10 SCV races, 9 are DCLs (90%).

It is well known from practical experience and from the literature [13, 26] that real programs contain many data races that users and developers do not consider important enough to report or to fix. Consequently, to put the previous numbers in context, we have to assume that there is a potentially sizable number of additional, unreported data races. Therefore, we can build the tree of Figure 4(a), which shows the frequency of each type of data race relative to its parent's. To visualize the frequency relative to all the race instances, Figure 4(b) shows a diagram where the area is proportional to the frequency of occurrence. Even if we do not know the actual number of unreported data races, the figure suggests that previous approaches that focus on data races as proxies for SCVs are insufficient.

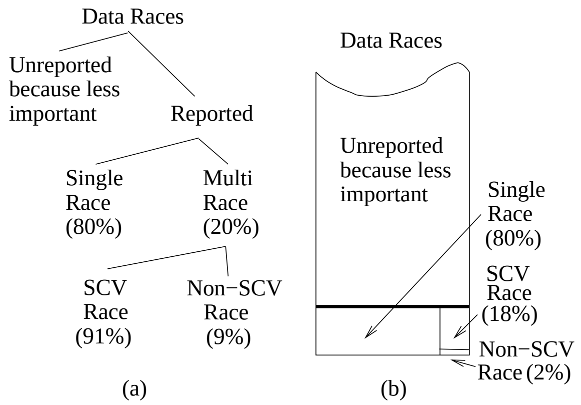


Figure 4. Relative frequency of data-race types.

The figure also shows why a special technique for SCV races is warranted: they comprise a substantial fraction of the reported data races, namely 18%. Importantly, they are very hard to debug, since current debuggers cannot reproduce them.

4. Vulcan: Detecting SC Violations

Our goal is to develop an approach to detect SCVs in relaxed-consistency machines that is highly precise. In addition, we want a solution that can deliver information to debug the SCV, uses no other input than the executable, and has a negligible execution overhead. Hence, we focus on a hardware-only solution to detect cycles of inter-thread data dependences at runtime.

The idea behind our approach, called *Vulcan*, is to rely on the cache coherence protocol to dynamically record the observed inter-thread data dependences, while checking 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 to minimize execution overhead.

4.1. Basic Algorithm to Detect Cycles

Figure 5(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.

To understand how Vulcan works, consider the dependence arrow of Figure 5(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 SCV. 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 5(c):

- 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 5(d). In the figure, for each of the two dependences, we use the algorithm of Figure 5(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 5(e). The first one is the Sequence 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 ∞.

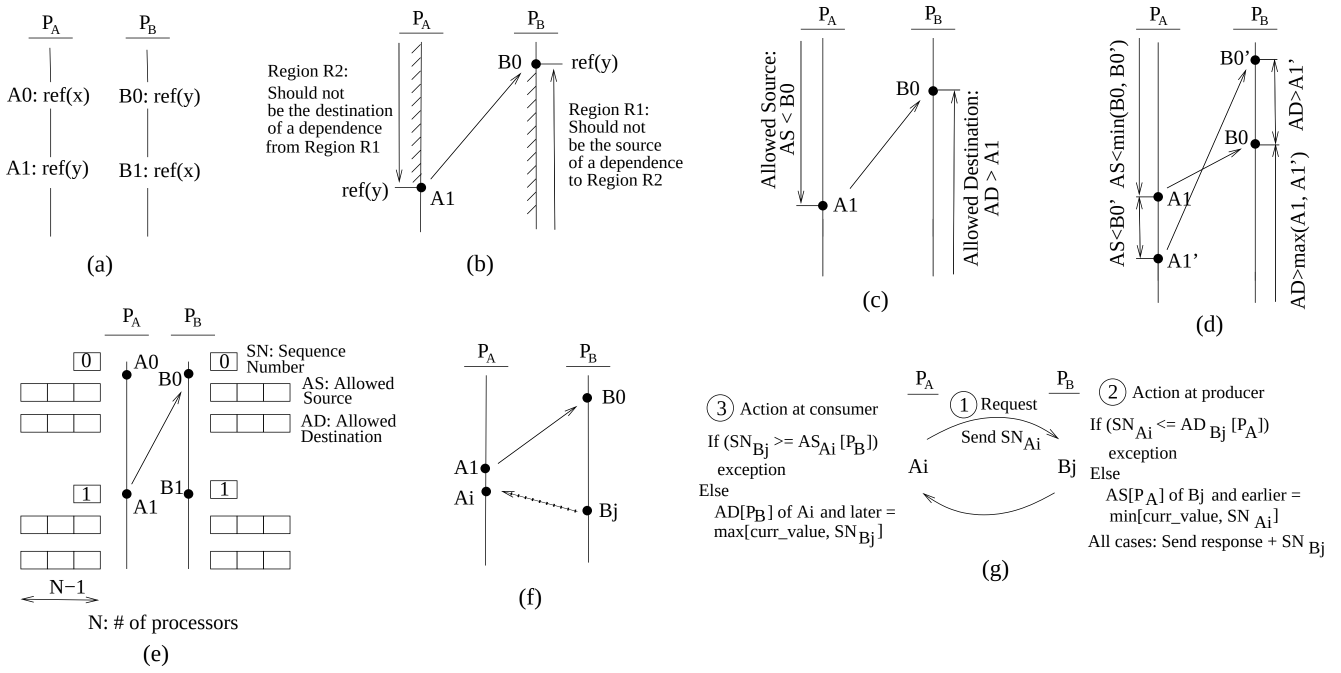

A0: ref(x)
A1: ref(y)
B0: ref(y)
B1: ref(x)
Region R2: Should not be the destination of a dependence from Region R1
Region R1: Should not be the source of a dependence to Region R2
Allowed Source: AS < B0
Allowed Destination: AD > A1
AS<B0' AS<min(B0, B0')
AD>A1' AD>max(A1, A1')
SN: Sequence Number
AS: Allowed Source
AD: Allowed Destination
N: # of processors
Action at consumer
If (SNBj >= ASAi [PB]) exception Else AD[PB] of Ai and later = max[curr_value, SNBj]
Request Send SNAi
Action at producer
If (SNAi <= AD Bj [PA]) exception Else AS[PA] of Bj and earlier = min[curr_value, SN Ai]
All cases: Send response + SN Bj

Figure 5. Basic algorithm to detect cycles.

These structures are updated in hardware when a new cross-processor dependence is created. The algorithm is shown in Figure 5(g), which refers to the example in Figure 5(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 5(g) shows that there are three steps in the creation of the *Bj→Ai* arrow. 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, hence an SCV 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 SCV has occurred. Consequently, an 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 causes an SCV. This provides valuable information for debugging the SCV. The exception at the processor that receives the response always occurs. The exception at the producer processor may not occur since, at send time, there may not be enough dependences for a cycle yet. In Section 5.5, we consider all the information that is available to debug the SCV.

4.2. Safe Accesses

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

- An access is *Safe* when no data dependence involving this access can cause an SCV any more. Otherwise, it is *Unsafe*. Vulcan can stop monitoring an access when it becomes Safe.

To find out when an access becomes Safe, let us define the *Performed Point* (PP) of a thread 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 the thread in program order have been performed. A load is performed when it has retired; a store is performed when it has retired and the cache has received the line and all the invalidation acknowledgments.

As a thread executes, its PP keeps advancing. When the PP reaches an access, it is clear that the access is completed. However, the access *may still* participate in an SCV and, therefore, be Unsafe. To see why, consider Figure 6(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 Unsafe for as long as P_A's PP has not reached the reference in their AD (*A1* in the example).

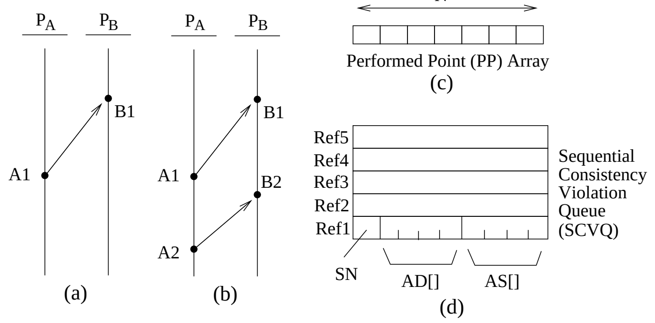

N
Performed Point (PP) Array
Ref5
Ref4
Ref3
Ref2
Ref1
SN
AD[]
AS[]
Sequential Consistency Violation Queue (SCVQ)

Figure 6. Understanding when an access is Safe.

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

• 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 Appendix 1].

As an example, consider Figure 6(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. After that, as soon as *A1* becomes Safe, all the accesses in P_B up to (but not including) *B2* 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 6(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, a processor only keeps the *SN*, *AD*, and *AS* information for its references that are Unsafe. Such information is kept in a per-processor FIFO hardware queue associated with the cache controller called *SC Violation Queue* (SCVQ) (Figure 6(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.

4.3. Detecting Dependences

When an SCV 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 SCV. [Proof in *Theorem 2* of Appendix 1].

For example, in Figure 7(a), arrow *1* could participate in an SCV, while arrow *2* cannot. 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.

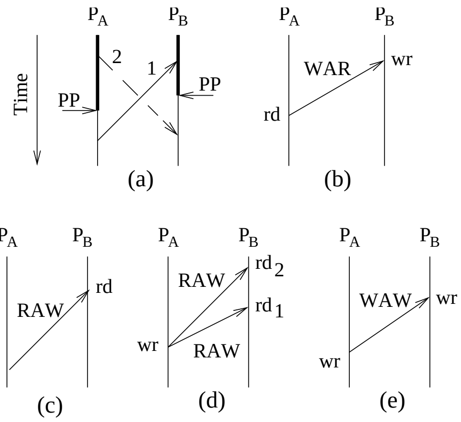


Figure 7. Inter-processor data dependences.

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 4.1: 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 7(b)-(e) show the three types of dependences possible: WAR, RAW, and WAW. Figure 7(b) shows a WAR. The write triggers Vulcan to search the other processors' SCVQs for accesses 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 *all* 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 7(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 7(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 writer's SCVQ entry. In the process, rd_1 will first set the *AS* of the write (and of P_A's prior accesses) to rd_1's *SN*; later, rd_2 will set them to rd_2's *SN*, which is lower.

Figure 7(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 forms 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.

We next show how we detect all the Unsafe dependences. The Appendix shows that:

• If Vulcan records all the Unsafe dependences, then it detects all the SCVs between processors. [Proof in *Theorem 3* of Appendix 1].

4.4. Leveraging the Coherence Protocol

To detect all the Unsafe dependences, Vulcan partially relies on piggybacking on the cache coherence protocol transactions. In this paper, 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, and multi-word cache lines. Without loss of generality, we describe our system using words (i.e., 32 bits) as the grain of processor accesses. We later consider finer-grained accesses such as bytes.

To understand how Vulcan uses the coherence protocol, this section starts by assuming single-word cache lines; Section 5 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 7 induces a coherence transaction in the network. Vulcan leverages such a transaction. The only exception is the second read (rd_2) in the RAW with out-of-order reads to the same address (Figure 7(d)). We describe this special case later.

As part of the coherence transaction, if the source reference is Unsafe (i.e., it is in an SCVQ), the Vulcan metadata is exchanged and operated upon. Specifically, on a processor read transaction in the network, the hardware searches the SCVQs that may contain the referenced address (we will see how we know this). In a given SCVQ, it tries to find the latest write to the address in program order. From the above discussion, at most one SCVQ can have writes. If a write is found, we have detected a RAW. The Vulcan metadata is exchanged (as part of the transaction) and operated upon.

On a processor write transaction in the network, the hardware searches the SCVQs that may contain the referenced address. In each SCVQ, the search tries to find the latest access to the address in program order and, if that is a read, also any preceding write. Vulcan looks for the latest accesses because they form the most conservative dependences. If we find any, we have detected a WAR or a WAW. The metadata is exchanged and operated upon. As part of the transaction, all the entries for the address in all SCVQs (except in the requesting processor) are invalidated.

The second read (rd_2) in the RAW with out-of-order reads of Figure 7(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 Network Access*, namely one exactly like a regular one (the SCVQs are searched and, if there is a hit, the Vulcan metadata is exchanged and operated on) except that no data is returned. Hence, when a load executes and finds that a later load to the same address has accessed the network, the hardware forces a metadata network access.

Vulcan's operation requires that, on a network 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. Consequently, Vulcan adds a per-processor bloom filter that encodes the addresses currently in the local SCVQ. If the address on the network hits in the filter, the SCVQ is searched. Section 5.3 presents a detailed design.

5. Vulcan Hardware Design

We present Vulcan's hardware structures: the coherence protocol and the SCVQs. We use a bus for the network. Section 7.2 summarizes the hardware needs for the configurations evaluated.

5.1. Supporting Multiple Words per Line

Detecting all the Unsafe dependences was easy with single-word cache lines because, conveniently, in all cross-thread dependences (Unsafe or otherwise, and except for RAWs with out-of-order read-read) the destination reference induces a coherence action in an MSI protocol (Figure 7). During the resulting bus access, if the dependence is Unsafe, processors exchange Vulcan metadata. Unfortunately, this is not the case with multi-word cache lines. As a processor misses on a word, other words are also brought into the cache. Consequently, some Unsafe dependences do not trigger coherence actions. Further, some coherence actions are caused by false sharing rather than by data dependences.

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

Let us use a plain line-based MSI coherence protocol using word accesses (for now). We assume that a bus transaction includes the address of the word accessed within the line. Vulcan adds two State bits per word in each line currently in the cache. These bits represent the word's *Vulcan-State* (or V-State). A word in the cache can be in one of three V-states: *CanWrite*, *CanRead*, and *NeedCheck*. Irrespective of the cache line state, a processor can write and read a *CanWrite* word in its cache without trying to exchange Vulcan metadata; it can only read a *CanRead* word without trying to exchange metadata; and it must try to exchange metadata at every access to a *NeedCheck* word. When needed, Vulcan metadata is piggybacked on the coherence bus transaction if the access induces one; otherwise, a Metadata bus accesses is initiated. These V-states are largely independent of the cache coherence state of the line. They follow rules when multiple caches have coherent copies of the word. Specifically, if one cache keeps the word in *CanWrite* state, then any other cache with the word must keep it in *NeedCheck* state. Also, if one cache keeps it in *CanRead* state, then any other cache can keep it in *CanRead* or *NeedCheck* state. Finally, the word may be in *NeedCheck* state in all of the cached copies.

Before describing how a word reaches each state, consider the (word) addresses of the accesses in an SCVQ. Typically, their corresponding line addresses are present in the local cache. However, there is one exception: when, after the access, the line was invalidated or displaced from the cache. In this case, the corresponding entries in the SCVQ have no V-state. In addition, when an invalidation is received, the SCVQ entry for the written word is cleared.

A word *w* in a line cached by a processor reaches the three V-states as follows:

• *CanWrite*: Either (i) the local processor was the last writer of *w* or, (ii) when the processor loaded *w* into its cache on a write miss to another word of the line, *w* was not in any other SCVQ (if the line was in another cache, it got invalidated). In addition, since any of these two events occurred, no other processor has (i) accessed *w*, or (ii) read-missed on another word in *w*'s line and loaded *w* as *CanRead*, or (iii) written *w*'s line. A *CanWrite* word may be in the local processor's SCVQ but not in other processors' SCVQs.

• *CanRead*: Either (i) the local processor has been involved in a dependence where the destination was a read of *w* (i.e., either the local processor wrote and then a remote one read, or a remote one wrote and then the local one read), or (ii) when the processor loaded *w* into its cache on a read miss to another word of the line, *w* was not in any other SCVQ. In addition, since any of these two events occurred, the local processor has not written *w* and no other processor has written to *w*'s line. A *CanRead* word may be in the SCVQs of the local and other processors.

• *NeedCheck*: When the local processor loaded *w* on a miss to another word of the line, *w* was in another processor's SCVQ. Since then, the local processor has not accessed *w* and no other processor has written to *w*'s line. A *NeedCheck* word may be in the SCVQs of the local and other processors.

We handle out-of-order read-read accesses to the same word like in Section 4.4: when a read executes and finds that a later read to the same address has already been sent to the bus, the hardware will eventually force a second bus access.

5.2. V-State Transitions for a Word

Figure 8 shows how the V-state of a word changes. For simplicity, we break the transitions into two figures. Figure 8(a) shows the transitions of the word as its line moves in and out of the cache, possibly due to accesses to other words in the same line; Figure 8(b) shows the transitions as the word is accessed inside the cache.

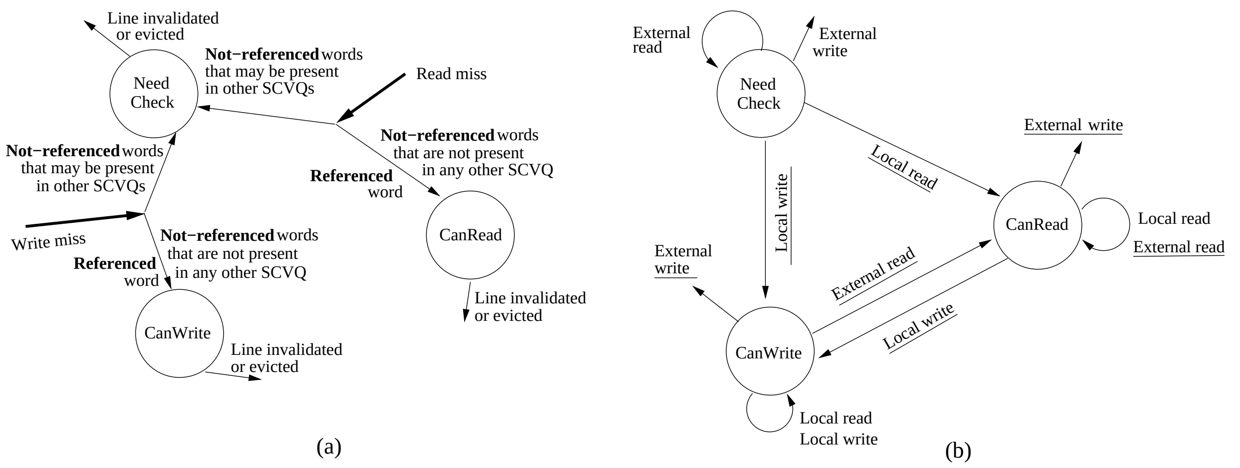

Line invalidated or evicted
Need Check
Not−referenced words that may be present in other SCVQs
Read miss
Not−referenced words that are not present in any other SCVQ
Referenced word
Not−referenced words that may be present in other SCVQs
CanRead
Write miss
Not−referenced words that are not present in any other SCVQ
Referenced word
Line invalidated or evicted
CanWrite
Line invalidated or evicted
(a)
External read
External write
Need Check
External write
Local read
Local write
CanRead
Local read
External read
External write
External read
Local write
CanWrite
Local read
Local write
(b)

Figure 8. V-state transitions for a word. In (b), the underlined transitions may require metadata exchange (only needed if the source of the dependence is Unsafe) and, therefore, need a bus access. Such access can reuse a coherence bus transaction.

Starting with Figure 8(a), as a processor brings-in a line on a read miss, the hardware operates on the Vulcan metadata of the *referenced* word as indicated before, recording any Unsafe dependence. Hence, the word is loaded into the cache as *CanRead*. The other words in the line (i.e., *not-referenced* words) are loaded as either *CanRead* — if their address is not in any of the other processor's SCVQs — or as *NeedCheck* otherwise. This functionality is supported by adding one control line in the bus for each word in a line. During the bus transaction, all the other processors also check the addresses of the not-referenced words in the line against their bloom filter. If any processor finds a match for a given word, it sets the control line for that word. If the control line for a particular word is not set by the end of the bus transaction, it means that no processor has the word in its SCVQ, and the word is loaded as *CanRead*. As a word is loaded as *CanRead*, any other cached copies of the word that were *CanWrite* transition to *CanRead*.

Hardware-prefetched lines work seamlessly. We apply the algorithm for not-referenced words to all the words in the line.

If the line is brought-in on a write miss, the state becomes *CanWrite* for the referenced word. For the other words, the bloom filters are checked as above and the state is set as *CanWrite* if no SCVQ has the address or *NeedCheck* otherwise. Other cached copies of the line are invalidated.

When a line is evicted from the cache or invalidated by an external write to any of its words, the V-states of all its words are lost.

In Figure 8(b), the word is being accessed. The transitions correspond to accesses to the word. The transitions underlined may require metadata exchange (only needed if the source of the dependence is Unsafe) and, therefore, need a bus access — which can reuse a coherence transaction. Consider a *CanWrite* word. The local processor can read and write it silently. An external read requires a transition to *CanRead* and attempts metadata exchange. Consider a *CanRead* word. A local read is silent. A local write brings the local state to *CanWrite* and induces a bus access to try to exchange metadata. All other copies of the line are invalidated. An external read keeps the local state as *CanRead* and may involve metadata exchange. Finally, in an *NeedCheck* word, a local read and write bring the word to *CanRead* and *CanWrite*, respectively, and induce a bus access to try to exchange metadata. An external read keeps the word in *NeedCheck*. In all states, an external write invalidates the line (and the corresponding SCVQ entry). It may involve metadata exchange if the state was *CanRead* or *CanWrite*.

Figure 9 shows two examples of processors *P1* and *P2* accessing a line with words *A* and *B*. The figures show the transitions in V-states and line states. For each access (e.g., *rd A* by *P2*), the figure shows the resulting local V-state of each of the two words (*CW*, *CR*, and *NC* mean *CanWrite*, *CanRead*, and *NeedCheck*, respectively), the resulting local line state (*D* and *S* mean Dirty and Shared Clean, respectively), and the type of bus request (*CO* and *ME* mean coherence and metadata request, respectively). For example, the first read in Figure 9(a) brings the line to *P2* in state *S* with both words as *CanRead*. This is a coherence request without metadata exchange. As we go down the access stream, some accesses cause bus requests with only metadata exchange. We assume all SCVQ entries stay unless they are invalidated. Figure 9(b) shows an access stream with false sharing. All accesses cause coherence-only bus requests.

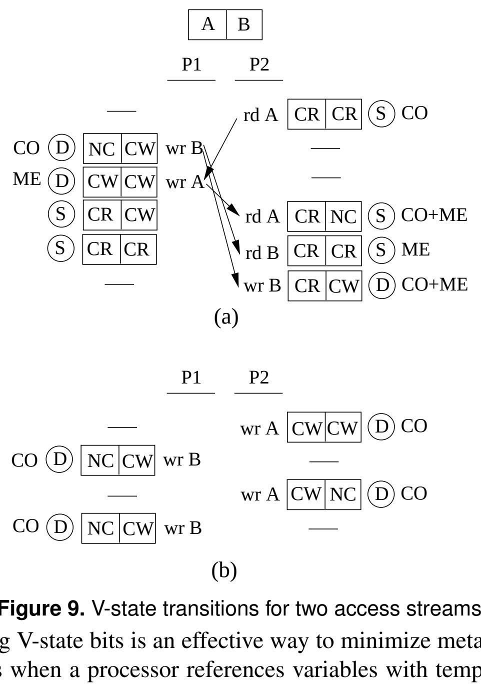


Figure 9. V-state transitions for two access streams.

Using V-state bits is an effective way to minimize metadata bus accesses when a processor references variables with temporal and spatial locality. Indeed, without V-state bits, all the words in the cache would effectively be in *NeedCheck* state, and every single access would require a metadata bus access. Unfortunately, V-state bits take space. Hence, a compromise that we employ is to keep

V-state bits *only* for lines that currently have at least one word in the local SCVQ. Since processor references have spatial and temporal locality, we are still likely to avoid many metadata bus accesses. When all the addresses of the words in the line leave the SCVQ (in addition to when the line is invalidated or evicted from the cache), the line's V-state information is discarded. A subsequent cache hit on the line initializes the V-state bits as *NeedCheck* for all the words (before the access). With this optimization, the V-state bits are stored in a hardware structure whose size is proportional to the maximum number of SCVQ entries rather than the number of lines in the L1 cache.

5.3. SCVQ Implementation

The SC Violation Queue (SCVQ) is a FIFO queue that contains the Vulcan metadata for Unsafe local loads and stores. Each entry contains the address loaded or stored, and the access' SN, AS, and AD. As a load or store enters the pipeline, an SCVQ entry is allocated, setting SN to the current value plus one, AS to ∞, and AD to the preceding access' AD. The AS and AD are updated later, when (1) the load or store executes, or (2) external accesses create dependences with the load or store, or other entries in the SCVQ.

Figure 10 shows the SCVQ. It stores the information in a FIFO circular queue. On a bus transaction, we need to look-up the SCVQ for an address match. Hence, we route the word addresses from the bus through a hash table and into the queue. With this design, it is easy to allocate and deallocate entries, and to find the entries that match bus transaction addresses. Finally, a write bus transaction that invalidates an SCVQ entry simply marks it as "empty".

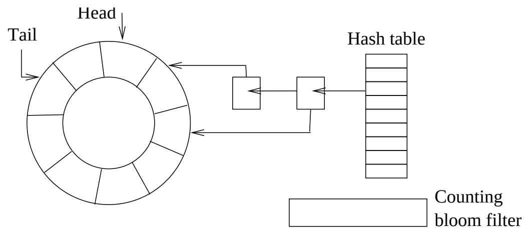


Figure 10. Implementation of the SC Violation Queue (SCVQ).

We want to minimize the number of useless SCVQ look-ups. However, we cannot rely on the cache snooper to filter them. This is because an SCVQ match may occur even if the corresponding line has been evicted from the cache. Hence, Vulcan uses a counting bloom filter [5] that hashes all the word addresses currently in the SCVQ. 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. Bus transactions check the filter for a match before initiating a hash-table access. Any resulting false positives do not affect correctness; false negatives do not occur.

Inserting or 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 or store completes and can be the source of an inter-thread dependence — in the meantime, the SCVQ entry is effectively not full. Removal can be done lazily, since at most it can induce false positive filter matches, which cause unnecessary SCVQ searches.

5.4. Granularity of V-State Bits

For most accurate SCV detection, the finest granularity of a program's accesses and the granularity of Vulcan's V-state bits have to be the same. Specifically, if a program loads or stores bytes, then Vulcan needs per-byte V-state bits — with per-word V-state bits, Vulcan may incur SCV false positives and false negatives.

To support byte accesses, Vulcan adds per-byte V-state bits to each line with at least one entry in the SCVQ. Individual entries in the SCVQ may refer to a byte or to a coarser access. Since the SCVQ bloom filter is looked-up by bus transactions accessing bytes or coarser data elements, Vulcan conservatively hashes word (rather than byte) addresses in the filter — at worst, it results in unnecessary SCVQ lookups. The transitions of Figure 8 operate on bytes or words depending on the granularity of the access. Specifically, on a byte access, when a line is brought into the cache (Figure 8(a)), the requested byte is searched in all of the SCVQs and loaded in the correct state; the other bytes of the line are loaded as *NeedCheck* or *CanRead/CanWrite* depending on whether their *word* address hits in the bloom filters. There are no additional filter lookups over a word transaction. Note that the design is such that, if the program only has word accesses, the per-byte V-state bits create no additional traffic over having only per-word V-state bits.

5.5. Information Available to Debug an SCV

Consider the cycle shown in Figure 2(b). There are two possible cases for when the SCV is detected. The first case is when one dependence arrow (e.g., *A1*→*B0*) is fully recorded when the source of the second dependence (*B1*) sends the response; the second case is when it is not, because both responders (*A1* and *B1*) respond concurrently. In the first case, the SCV is detected at both the source (*B1*) and destination (*A0*) of the second dependence; in the second case, it is detected at the destinations of the two dependences (*A0* and *B0*). In either case, when each processor detects the SCV, it raises an exception.

The information that is available to the debugger in the interrupted processor at the destination of the dependence is the address being accessed, the instruction's PC and, depending on the protocol implementation, the ID of the sender processor. If the destination reference is a read, the exception gets the precise processor state; if it is write, it is not generally possible to get the precise state at the reference — the reason is that the write is in the write buffer and later operations may have already retired and completed. The information available to the debugger in the interrupted processor at the source of the dependence is the address accessed and the ID of the requesting processor. The instruction's PC is unavailable — unless we augment the SCVQ with PCs. The exception in the source processor is not precise because newer instructions may have finished. Finally, the debugger can also inspect the Vulcan metadata of all the Unsafe requests in the two processors, to provide more information.

6. Limitations of the Current Vulcan Design

The current Vulcan design has some limitations. The first one is that it focuses on cycles involving only two processors. In practice, this is not a major limitation because cycles involving more processors are much rarer — they need the overlapping of three or more data races. Much of the related work also focuses on two-processor interactions only (e.g., [6, 8, 10]). We could extend Vulcan to handle several-processor cycles by propagating the *AS/AD* information along the dependence arrows, instead of just sending *SN*.

A second limitation is that the current design does not consider speculative loads from mispredicted branch paths. In a real system, these loads cannot generate SCVs. However, to be able to take them into account, we would need to change Vulcan. For example, the hardware may have to delay performing metadata updates until the load becomes non-speculative. However, Vulcan supports hardware prefetches and within-processor load forwarding.

Vulcan is not concerned with the impact of compiler transformations on SCVs. It simply takes the executable that the compiler provides to the hardware and reports SCVs due to hardware-initiated reference reordering. Similarly, since Vulcan is a dynamic scheme, it only provides information for the actual performed runs.

We discussed in Section 5.4 that the finest granularity of program accesses and of Vulcan's V-state bits have to be the same — otherwise, both SCV false positives and false negatives may occur. Finally, the SCVQs need to be large enough to hold all the Unsafe accesses. If they are not and they have to drop some of these accesses, then SCV false negatives may occur.

Overall, within these constraints (two-processor cycles only, no misspeculated loads, and no compiler effects) and with appropriate hardware structures (correct grain of V-state bits and large-enough SCVQs), Vulcan has neither false positives nor false negatives.

Finally, our Vulcan design in a snoopy protocol with all-to-all hardware structures may not scale well to large numbers of processors. However, this is not a major limitation. First, our evaluation shows that Vulcan scales well until at least 8 processors (and we did not explore beyond). Also, it is well known that runs with few processors are typically enough to find concurrency bugs [22].

7. Evaluation

Our goal is to (1) validate Vulcan's effectiveness in detecting SCVs, (2) determine the size of its hardware structures, and (3) assess its overhead in terms of network traffic and execution time.

7.1. Experimental Setup

We model Vulcan's architecture using cycle-level execution-driven simulations. We use the SESC simulator [28] to model a multicore with a variety of configurations: four or eight out-of-order cores with 2- or 4-issue wide pipelines and supporting the Release Consistency (RC) or Processor Consistency (PC) memory models. They have a simple cache hierarchy composed of a private L1 cache and a shared L2 cache. Table 2 shows the architecture parameters. When there is a choice, the values in bold are the default ones. In most of the evaluation, we use per-word V-state bits; in the last part, we use per-byte V-state bits.

Architecture	Chip multiprocessor with **4** or 8 cores.
Core pipeline	Out-of-order; 2.0GHz; **2-issue** or 4-issue.
ROB size	**32**, 64, 128, or 256 entries.
Consistency	**Release (RC)** or Processor (PC) consistency.
Private L1 cache	32KB WB, 4-way asso., 2-cycle round trip.
Shared L2 cache	1MB WB, 8-way asso., 20-cycle round trip.
Cache line size	**32B** or 4B.
Coherence	Snoopy MSI protocol; 1.0GHz 16B-wide bus.
Round-trip lat.	L1-L1: 38 cyc; processor-memory: 500 cyc.
Vulcan parameters	SCVQ: 256 entries; SN, AD[i], AS[i]: 4B each. Bloom filter: 128B with 2-bit counts, H3 hash. **Word** or byte V-state bits for lines in SCVQ.

Table 2. Multicore architectures evaluated.

We use three sets of applications for the evaluation (Table 3). The first set has implementations of concurrent data structures and mutual exclusion algorithms that have SCVs. They are taken from [6, 8]. The second set has some reported SCV bugs from open source libraries. The last set has 8 codes from SPLASH-2. The first two sets have known SCVs and are used to evaluate Vulcan's effectiveness. The last set has lengthy applications, supposedly free of SCVs, and is used to estimate Vulcan's overheads.

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 a fence [3].
	Crypt_util from glibc	Small table initialization code needs a fence [1].
	DCL bug	Kernel using double checked locking without fences.
Full Apps	SPLASH-2	8 programs form SPLASH-2.

Table 3. Applications analyzed.

7.2. Hardware Requirements

Vulcan adds to each core the following hardware: (1) SCVQ circular queue with its hash table, (2) SCVQ bloom filter, (3) V-state bits in the lines with at least one entry in the SCVQ, and (4) Performed Point array. For the default parameters in Table 2, in a 4-core chip, the storage requirements are about 8448B, 128B, 512B, and 16B, respectively, or a total of 9KB per core. For a machine with *N* cores, the overhead per core can be shown to be (2052*N + 896) bytes. This means that, in an 8-core chip, the overhead is 17KB per core.

If we want to support byte-level accesses, we need per-byte V-state bits for each line with at least one entry in the SCVQ. Moreover, each SCVQ entry needs a 2-bit longer address and 2 bits to denote whether the reference was to a byte, half-word, or word. The SCVQ bloom filter still conservatively hashes word addresses. All this support only adds 1.7KB more Vulcan storage overhead per core, irrespective of the number of cores in the machine.

7.3. SC Violation Detection Ability

To test Vulcan's SCV 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. For each application, we report, over all the runs, the number of *unique* and *total* SCVs observed. This information is shown in Table 4, for cache lines of 4 and 32 bytes, and for RC and PC memory models. For SPLASH-2, the table only shows fmm because Vulcan finds no SCV in the other SPLASH-2 codes.

Under RC (Columns 4 and 5), Vulcan detects SCVs in all of these codes (except in two codes with 4B lines). Under PC (Columns 6 and 7), Vulcan finds slightly fewer unique SCVs, and none in Lazylist or Snark. This is because PC is stricter than RC, and some SCVs may be impossible or less likely to occur. Also, the number of SCVs found changes with the line size. This shows that this bug is highly dependent on the timing of events.

Overall, we find that Vulcan is very effective at finding SCVs in these two different memory models. With more runs, new interleavings may occur and Vulcan may find more SCVs.

Finally and most importantly, Vulcan *finds three new, previously unreported SC violation bugs* in the codes in bold in Table 4: one in Pthread_cancel, one in Crypt_util, and one in fmm (which appears as three unique SCVs). We discuss them next.

Appl.	Line Size (B)	# of Runs	# of SC Violations Found			
			Under RC		Under PC	
			Uniq.	Total	Uniq.	Total
Dekker	4	100	1	1982	1	1784
	32	100	1	224	1	518
Lazylist	4	100	0	0	0	0
	32	100	1	150	0	0
Snark	4	100	1	745	0	0
	32	100	1	1467	0	0
Harris	4	100	0	0	1	2
	32	100	1	18	1	2
Pthread _cancel	4	100	2	298	1	104
	32	100	2	142	1	400
Crypt _util	4	100	2	564	1	228
	32	100	2	130	1	800
DCL	4	100	2	648	1	600
	32	100	1	2	1	491
fmm	4	5	1	2	3	14
	32	5	3	18	0	0

Table 4. SC violations found in various applications. Vulcan found three new SC violations in the codes in bold.

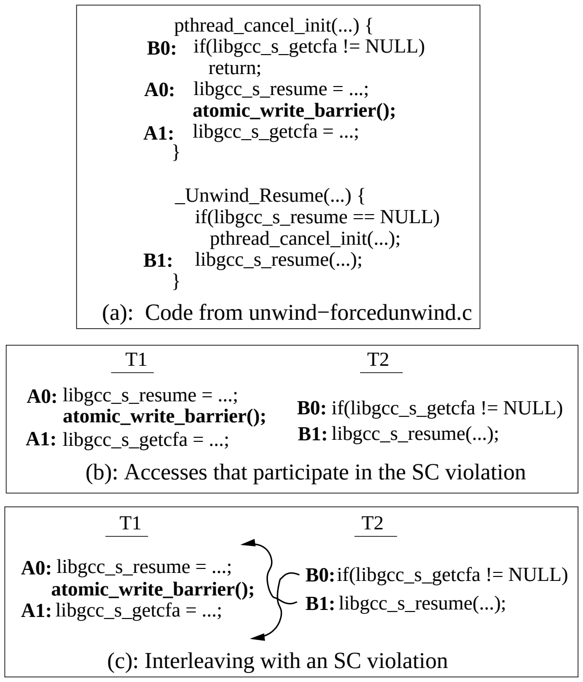
7.4. Three New SC Violation Bugs Found

• New SC Violation in the Pthread Library

One of the SCVs in the *Pthread_cancel* kernel of Table 4 is Bug ID 2644 in the Redhat bug database [3], which has been fixed by the developers. After running Vulcan, we found a *new SC violation even in the bug fix*. We reported the new bug and its fix to the developers, who have recently implemented the fix.

Figure 11 shows the bug in the original bug fix. Figure 11(a) shows the *pthread_cancel_init* and *_Unwind_Resume* functions, together with the fence (write barrier) that the developers inserted in an attempt to fix the bug. Assume that thread T1 is in *pthread_cancel_init*, and about to initialize function pointers *libgcc_s_resume* (in A0) and *libgcc_s_gtecfa* (in A1). Before it does so, 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). However, due to an SCV, *libgcc_s_resume* is still uninitialized and the program crashes.

The references involved and the fence are shown in Figure 11(b). This code is the same as Figure 1(a) except for the fence. Unfortunately, the fence only prevents the A0-A1 reorder. In an RC (or PowerPC) memory model, B0 and B1 can *effectively* get reordered as in Figure 11(c), causing a cycle. Specifically, the condition in B0 is predicted true by the branch predictor (although it is currently false) and B1 is executed before A0. After A0 and A1 execute, the B0 branch resolves, confirming that B1 is in the correct path. However, B1 used the old value and the code crashes. To fix this, we also add a fence between B0 and B1.



```
        pthread_cancel_init(...) {
B0:       if(libgcc_s_getcfa != NULL)
            return;
A0:       libgcc_s_resume = ...;
          atomic_write_barrier();
A1:       libgcc_s_getcfa = ...;
        }

        _Unwind_Resume(...) {
          if(libgcc_s_resume == NULL)
            pthread_cancel_init(...);
B1:       libgcc_s_resume(...);
        }
```

(a): Code from unwind−forcedunwind.c

```
              T1                                   T2

A0: libgcc_s_resume = ...;            B0: if(libgcc_s_getcfa != NULL)
    atomic_write_barrier();           B1: libgcc_s_resume(...);
A1: libgcc_s_getcfa = ...;
```

(b): Accesses that participate in the SC violation

```
              T1                                   T2

A0: libgcc_s_resume = ...;            B0: if(libgcc_s_getcfa != NULL)
    atomic_write_barrier();           B1: libgcc_s_resume(...);
A1: libgcc_s_getcfa = ...;
```

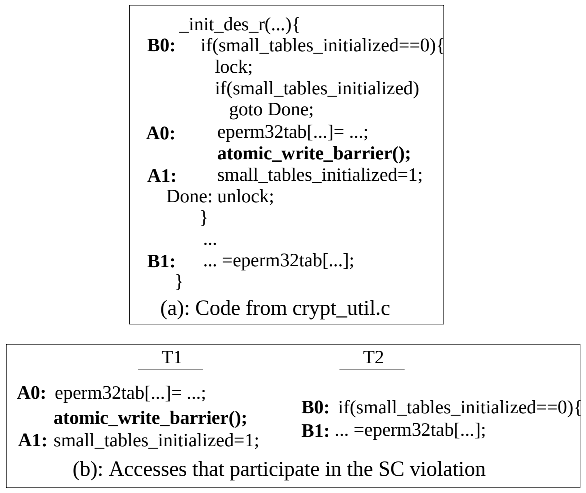
(c): Interleaving with an SC violation

Figure 11. New SC violation found in the glibc pthread library.

• New SC Violation in the Crypt Library

A similar situation occurs for *Crypt_util*. One of its SCVs in Table 4 is Bug ID 11449 in the database [1], which had also been incorrectly fixed by the developers. After running Vulcan, we found a new SCV in the bug fix. We reported the new bug and its fix to the developers. They declined to fix it because the bug also only happens in memory models more relaxed than Intel's x86 and the cryptography library is used little.

Figure 12(a) shows the buggy code of function *__init_des_r*, which uses DCL to initialize shared tables, and the fence that the developers added to fix the bug. Assume that thread T1 enters the function, grabs the lock and is about to initialize table *eperm32tab* (in A0) and then set *small_tables- _initialized* (in A1). Thread T2 enters the function, finds *small_tables_initialized* set (in B0) and uses *eperm32tab* (in B1). Unfortunately, *eperm32tab* is still uninitialized due to the SCV.



```
       _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[...]= ...;             B0: if(small_tables_initialized==0){
    atomic_write_barrier();           B1: ... =eperm32tab[...];
A1: small_tables_initialized=1;
```

(b): Accesses that participate in the SC violation

Figure 12. New SC violation found in the glibc crypt library.

The references involved and the fence are shown in Figure 12(b). The code is similar to the one in Figure 11(b). We need another fence between B0 and B1.

• New SC Violation in fmm from SPLASH-2

Vulcan finds three new SCVs 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 13. Inside the *SetColleagues* 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 the pattern of Figure 1, and an SCV 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.

```
        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 13. New SC violation found in fmm from SPLASH-2.

This SCV 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, this is a serious bug because the programmer can hardly reason about the bug's impact on the code. 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.

7.5. SCVQ Size and Sensitivity

To size the SCVQ, we need to know the number of Unsafe accesses that individual processors maintain. Consequently, we count the average and maximum number of Unsafe accesses per processor over time. We use only SPLASH-2 applications because the others are too small to provide useful information. For our measurements, we take a sample every time a memory operation is issued. We additionally count the average and maximum number of pending accesses. These are loads and stores that have been issued but not yet completed, and are a strict subset of Unsafe accesses — an access remains Unsafe at least while pending and often beyond that. Figure 14 shows the results for each application.

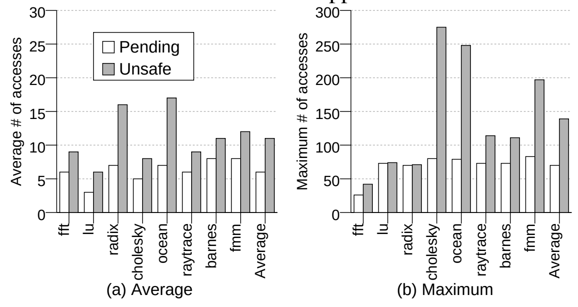


Figure 14. Number of pending and Unsafe accesses.

The average number of Unsafe accesses ranges from 6 to 17 (Figure 14(a)). This is a small number, and is about double of the average number of pending accesses. However, accesses are typically bursty and the maximum number of Unsafe accesses is higher. Across applications, it ranges from 40 to 270 (Figure 14(b)). If we average out all the codes, the number is about 140, which is also about double of the maximum number of pending accesses.

Overall, to be conservative, we size the SCVQ with 256 entries. Most of the time, only about 10 or so entries are in use. In one application, namely cholesky, there are 170 times in the execution of the 147-million memory-access program when we need more than 256 entries. Hence, we have rerun the program with a 270-entry SCVQ, which is large enough, and found no SCVs either.

We now measure how the number of Unsafe and pending accesses changes with the ROB size and processor issue width. This is shown in Figure 15, which plots the average across all SPLASH-2 codes. For each ROB size and issue width, we show the average and maximum number of pending and Unsafe accesses. The number on top of the maximum Unsafe bars is the number of SCVQ overflows, as a percentage of total instructions. We see that, for our default issue width (Figure 15(a)), changes in the ROB size have negligible impact. For 4-issue cores (Figure 15(b)), if the ROB reaches 128 entries or more, the SCVQ starts to overflow.

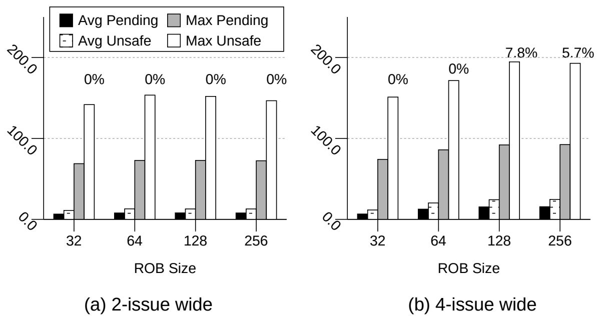


Figure 15. Pending and Unsafe accesses for different ROB sizes and issue widths.

7.6. Network Traffic & Execution Overhead

Vulcan's execution overhead comes from the additional bus traffic that it induces. This traffic has two sources: (i) the information that Vulcan piggybacks on some of the ordinary coherence transactions on the bus and (ii) the Metadata bus accesses that it induces (Section 4.4). In both cases, Vulcan sends a Sequence Number in the request (4 bytes), and both a Sequence Number and a Performed Point in the response (8 bytes).

To see the magnitude of this traffic, Figure 16 breaks down the total bytes of traffic in the bus for each application. We run the experiments for both 4-core and 8-core systems. The categories are: traffic in a Vulcan-free execution (*No Vulcan*), traffic piggybacked by Vulcan on the normal coherence (*Piggybacked*), and traffic in Metadata bus accesses (*Extra*). We see that Vulcan's effect is modest: on average for 4 cores, *Piggybacked* accounts for 9% of the traffic and *Extra* for 12%. For 8 cores, the result is similar.

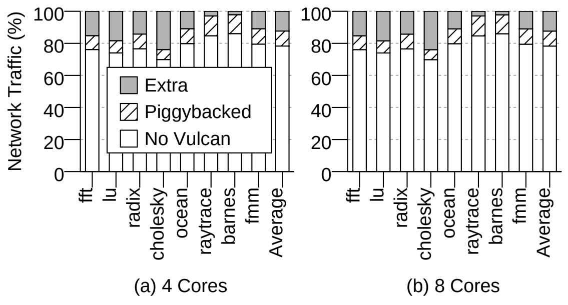


Figure 16. Breakdown of total bus traffic in bytes.

Given the bus parameters of Table 2, we assume that the additional bytes piggybacked by Vulcan on a coherence transaction do not increase the bus occupancy cycles of the transaction. However, for Metadata bus accesses, we assume bus occupancies of 2 bus cycles for request and 2 for reply. The contention induced by these accesses causes Vulcan's execution overhead.

Tables 5 and 6 show Vulcan's execution overhead for 4 and 8 core systems, respectively. Each table shows the execution overhead with word and byte granularity for V-state bits. For each core count and V-state granularity, the tables show the number of bus accesses, the fraction of those that are Metadata bus accesses, and the increase in the program's execution time due to Vulcan.

Appl.	Word Granularity			Byte Granularity		
	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)
fft	0.4	32.4	4.9	0.4	32.6	4.9
lu	1.2	34.4	3.8	1.2	34.4	3.8
radix	2.0	32.5	0.7	2.0	32.6	0.7
chole.	34.4	38.9	8.1	34.4	38.9	8.1
ocean	21.5	26.7	5.5	21.5	26.7	5.5
raytr.	3.1	8.8	4.2	3.6	19.8	6.7
barnes	30.7	6.9	2.7	30.9	6.9	2.7
fmm	19.2	25.8	2.6	19.2	25.8	2.6
Avg.	14.1	25.8	4.1	14.2	27.2	4.4

Table 5. Vulcan's execution overhead for 4 cores.

Appl.	Word Granularity			Byte Granularity		
	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)	Tot. Bus Acc. (Mil.)	Meta. Bus Acc. (%)	Exec. Time Over. (%)
fft	0.4	31.8	9.5	0.4	31.9	9.5
lu	1.2	35.3	3.6	1.2	35.3	3.6
radix	2.1	33.0	1.4	2.1	33.0	1.4
chole.	35.6	38.4	9.0	35.7	38.5	9.0
ocean	21.6	27.7	12.3	21.7	27.6	12.3
raytr.	3.6	9.6	4.4	4.0	19.0	6.9
barnes	35.0	6.5	2.8	35.1	6.3	2.7
fmm	19.4	26.0	2.8	19.4	26.0	2.8
Avg.	14.9	26.0	5.7	14.9	27.2	6.0

Table 6. Vulcan's execution overhead for 8 cores.

The tables show that Metadata bus accesses account for an average of 26-27% of the bus accesses, and that such fraction does not change much with the core count. Importantly, Vulcan's execution time overhead is small. On average for word granularity, it is 4.1% for 4-core systems and 5.7% for 8-core systems.

When Vulcan supports V-state byte granularity, the overhead increases in the applications with a non-negligible fraction of byte accesses. For the applications considered, only Raytrace is in this class. As a result, in Raytrace, the number of Metadata bus accesses increases and the execution time overhead increases a modest 2.5 percentage points, as we go from word to byte granularity for both processor counts. For the other codes, since they reference mostly words rather than bytes, Vulcan's execution behaves as if it had word- rather than byte-granularity V-state bits. On average for all the applications, the execution overhead with byte-granularity V-state bits is 4.4% for 4-core systems and 6.0% for 8-core systems.

Overall, we conclude that Vulcan's execution overhead is small enough to allow on-the-fly use — both with word- and byte-granularity V-state bits. In addition, the overhead scales nicely from 4- to 8-core systems.

8. Other Related Work

There is related work in architecture, compilation, testing, and hardware verification. In architecture, the most related work is Conflict Ordering (CO) by Lin *et al.* [21], which is a technique to support SC in a relaxed-consistency machine. CO is also based on identifying Shasha's delay sets [30] in hardware. At a high level, CO and Vulcan differ in that their goals are different: Vulcan focuses on identifying SCVs, while CO focuses on supporting SC. However, Vulcan could be extended to support SC when an upcoming SCV is suspected, and CO could stop execution when an SCV is possible. Hence, at a deeper level, CO and Vulcan are similar in that they both attempt to identify race cycles.

CO's key contribution is to use information about pending accesses in the directory module to avert cycles. Unfortunately, CO requires introducing stalls in processor requests. Specifically, there are two stall types: write- and read-induced. Write-induced stalls occur when the write *W* that is about to retire misses in the cache. At that point, the next read or write cannot retire until *W* goes to the directory, leaves its address there, and brings back the list of pending writes (write-list). This stall cannot be eliminated with exclusive prefetching. Read-induced stalls occur when a speculative read *R* misses in the cache. When *R* reaches the ROB head, *R* has to perform a directory access again, to obtain the write-list. Only when the write-list returns can *R* retire and allow subsequent reads and writes to retire. Again, this cannot be fixed by prefetching.

CO also differs from Vulcan in that, to detect cycles, it compares line addresses rather than word or byte addresses. This causes false positives. Luckily, false positives simply cause stalls — although this approach would not work to debug SCVs like Vulcan. If, instead, CO compared word addresses, then a processor accessing multiple words of the same line in sequence would have to make multiple directory accesses to deposit the addresses of all the words.

There are compiler techniques to identify race cycles and put fences (e.g., [14, 17, 19, 31]). They are conservative because they only use static information, and typically cause large slowdowns. Lin *et al.* [20] can hide some of the resulting fence delay with architectural support. Duan *et al.* [12] use a race detector to construct a graph of races dynamically. Then, off-line, they traverse the graph to find potential SCVs. Vulcan differs in that: (1) it is an on-the-fly scheme, while Duan's SCV detection is off-line; (2) it needs no software support; and (3) it has no false positives, while Duan's scheme may point to SCVs that never occur.

The software testing community has proposed static and off-line techniques to check for SCVs (e.g., [6, 7, 8]). While promising, these techniques are not designed for on-the-fly SCV detection in large codes with negligible overhead. The hardware verification community has designed techniques to verify if a memory system hardware is correctly implemented (e.g., [10, 11, 25]). While related, these works have a different goal: we focus on debugging software as it runs on a relaxed-consistent machine; they focus on verifying that the hardware correctly implements a memory model.

9. Conclusion

This paper proposed Vulcan, the first hardware scheme to precisely detect SCVs at runtime, in programs running on a relaxed-consistency machine. Vulcan uses cache coherence protocol transactions to dynamically detect cycles in memory access orders across threads. When a cycle is about to occur, an exception is triggered, providing information to debug the SCV. For the conditions considered in this paper and with enough hardware, Vulcan suffers neither false positives nor false negatives. It induces negligible execution overhead, requires no software help, and only takes as input the program executable. Our results showed that Vulcan detected *three new SCV bugs* in popular codes: Pthread and Crypt libraries, and fmm from SPLASH-2. Vulcan's negligible execution overhead makes it suitable for on-the-fly use.

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] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *ISCA*, June 2009.

[5] F. Bonomi et al. An improved construction for counting Bloom filters. In *Ann. Euro. Symp. on Algo.*, September 2006.

[6] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *PLDI*, June 2007.

[7] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, July 2008.

[8] J. Burnim, K. Sen, and C. 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.

[9] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *ISCA*, June 2007.

[10] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *HPCA*, February 2008.

[11] A. Deorio et al. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *HPCA*, February 2009.

[12] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew. Detecting and eliminating potential violations of sequential consistency for concurrent C/C++ programs. In *CGO*, March 2009.

[13] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, February 2010.

[14] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *ICS*, June 2003.

[15] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *SPAA*, July 1991.

[16] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, May 1999.

[17] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jour. Paral. Dist. Comp.*, Nov 1996.

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

[19] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comp.*, August 2001.

[20] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *PACT*, September 2010.

[21] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient sequential consistency via conflict ordering. In *ASPLOS*, March 2012.

[22] S. Lu et al. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, March 2008.

[23] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, June 2010.

[24] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A simple and efficient memory model for concurrent programming languages. In *PLDI*, June 2010.

[25] A. Meixner and D. J. Sorin. Dynamic verification of sequential consistency. In *ISCA*, June 2005.

[26] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, June 2007.

[27] S. Rajamani et al. ISOLATOR: Dynamically ensuring isolation in concurrent programs. In *ASPLOS*, March 2009.

[28] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[29] D. C. Schmidt and T. Harrison. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. In *Patt. Lang. of Prog. Design*, 1996.

[30] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM TOPLAS*, April 1988.

[31] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent Java programs. In *PPoPP*, June 2005.

[32] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA*, June 2007.

[33] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, June 2009.

Appendix 1: Correctness Proofs

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: Recall that C_i becomes Safe as soon as it cannot participate in an SCV anymore. Assume that C_i can participate in an SCV with another access of P_C: either an earlier one C_{i-m} (Case 1) or a later one C_{i+m} (Case 2) in program order (Figure 17).

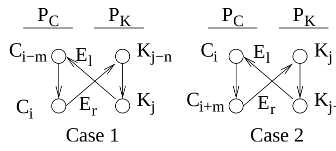


Figure 17. Possible cases for SCV.

Case 1: Consider two situations. In the first one, edge E_r occurs first. Although C_i has executed, it can still participate in an SCV for as long as P_C's previous accesses (C_{i-m} where $1 \leq m \leq i$) are not performed — since such accesses can still be the destination of an E_l edge. Hence, when $PP[P_C] \geq SN_{C_i}$, then C_i is Safe. The second situation is when edge E_l occurs first. C_i is not Safe until all of the P_K accesses up to K_j (K_{j-n} where $0 \leq n \leq j$) are performed, without consuming an edge from C_i. Note that the allowed destinations of C_i are the accesses after the E_l source K_j ($AD_{C_i}[P_K] = K_j$). The E_l edge can point to any P_C access preceding C_i. Hence, C_i is only Safe when it and all the previous accesses in P_C are performed, and all the accesses in P_K up to and including $AD_{C_i}[P_K]$ are performed. Hence, the Safe condition in Case 1 is ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$).

Case 2: There are two situations. In the first one, edge E_l occurs first. Although C_i has executed, it can still participate in an SCV for as long as its disallowed destinations in P_K (K_{j+n} and earlier) have not performed — since such accesses can be the destination of an E_r edge. Hence, when $AD_{C_i}[P_K] \leq PP[P_K]$, then C_i is Safe. The second situation is when E_r occurs first. In this case, when C_i performs, we know whether it creates a cycle with E_r. Hence, C_i is Safe when $SN_{C_i} \leq PP[P_C]$. Overall, the Safe condition in Case 2 is the same as Case 1, namely ($SN_{C_i} \leq PP[P_C]$) and ($AD_{C_i}[P_K] \leq PP[P_K]$).

Generalizing to all the processors, 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 their destination processors.
Proof: This is proved by contradiction. Assume that one of the dependences has a source that is Safe (with respect to the destination processor) and it forms an SCV cycle with another dependence whose source is Unsafe (with respect to the destination processor). According to the definition of a Safe access, once an access becomes Safe (with respect to a processor), no dependence from this access to an access of that other processor can cause an SCV. This contradicts our previous assumption and proves the theorem.

Theorem 3: If Vulcan records all the Unsafe dependences, then it detects all the SCVs between processors.
Proof: Referring to Figure 17, an access C_i can participate in an SCV with one of P_C's earlier accesses (Case 1) or one of the later accesses (Case 2). Without loss of generality, assume that, of the two dependence edges in the cycle, the edge at C_i is formed the latest. We now show that, when that edge is formed, C_i has all the information that it needs to detect the SCV.

In Case 1, the information that C_i needs to keep is the sources of the dependences pointing *to* any of the P_C accesses before C_i. More specifically, it needs to keep the maximum *SN* of such sources. With this information, it cannot miss a cycle when the edge at C_i occurs. But this is precisely the information in AD_{C_i}.

In Case 2, the information that C_i needs to keep is the destinations of the dependences pointing *from* any of the P_C accesses after C_i. More specifically, it needs to keep the minimum *SN* of such destinations. With it, C_i will not miss a cycle when the edge at C_i occurs. But this is precisely the information in AS_{C_i}.

Overall, if Vulcan records all the sources and destinations of the dependences (with AD and AS), it can find all the SCVs. Moreover, Theorem 2 proves that SCVs occur only among Unsafe dependences. Hence, if Vulcan records all the Unsafe dependences, then it detects all the SCVs.