Volition: Scalable and Precise Sequential Consistency Violation Detection

Xuehai Qian, Josep Torrellas
University of Illinois, USA
{xqian2, torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

Benjamin Sahelices
Universidad de Valladolid, Spain
benja@infor.uva.es

Depei Qian
Beihang University, China
depeiq@buaa.edu.cn

Abstract

Sequential Consistency (SC) is the most intuitive memory model, and SC Violations (SCVs) produce unintuitive, typically incorrect executions. Most prior SCV detection schemes have used data races as proxies for SCVs, which is highly imprecise. Other schemes that have targeted data-race cycles are either too conservative or are designed only for two-processor cycles and snoopy-based systems.

This paper presents *Volition*, the first hardware scheme that detects SCVs in a relaxed-consistency machine precisely, in a scalable manner, and for an arbitrary number of processors in the cycle. Volition 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. Volition can be used in both directory- and snoopy-based coherence protocols. Our simulations of Volition in a 64-processor multicore with directory-based coherence running SPLASH-2 and Parsec programs shows that Volition induces negligible traffic and execution overhead. In addition, it can detect SCVs with several processors. Volition is suitable for on-the-fly use.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors) - Parallel Processors

Keywords Memory Consistency, Sequential Consistency, Parallel Programming, Shared-Memory Multiprocessors

1. Introduction

When programmers write and debug applications with shared-memory threads, they intuitively assume the Sequential Consistency (SC) model. SC requires that the memory operations of a program appear to execute in some global sequence, as if the threads where multiplexed on a uniprocessor [17]. In practice, however, processors and memory systems overlap, pipeline, and reorder the memory accesses of threads. As a result, the execution of a parallel program can violate SC.

As an example, consider Figure 1(a). Processor P_0 initializes variable *p* and then sets flag *OK*; later, P_1 tests *OK* and, if it is set, uses *p*. While the interleaving in Figure 1(a) produces the expected results, the interleaving in Figure 1(b) does not. Here, while the two writes in A0 and A1 are retired in order, they complete out of order: the first one after B0 and B1, and the second one before B0 and B1. In this interleaving, P_1 ends up using an unitialized *p*. This order is an SC Violation (SCV).

While this example is trivial, SCVs often appear under subtle thread interleavings and timing conditions, even in popular codes. For example, Muzahid et al. [24] discovered SCVs in the Pthread and Crypt libraries of glibc. SCVs are often found in double-checked locking constructs [27], some synchronization libraries, and code for lock-free data structures. In Section 2, we show a typical example of SCV.

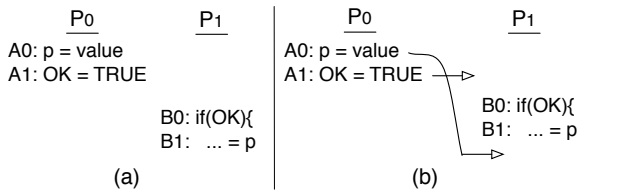

Figure 1. Example of an SC violation.

From the hardware perspective, an SCV occurs only when multiple conditions are met. First, there needs to be two or more data races — e.g., the races on variables *p* and *OK* in Figure 1. Second, these races must overlap in time. Finally, the order of the references in these races has to form a cycle at runtime [28].

Specifically, for two threads, an SCV requires a pattern like that in Figure 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 references in these two racing pairs have to form a cycle as shown in Figure 2(b) — where we have arbitrarily picked reads and writes. Specifically, A1 must occur before B0, and B1 must occur before A0. This what happens in Figure 1(b), where *y* is *OK* and *x* is *p*.

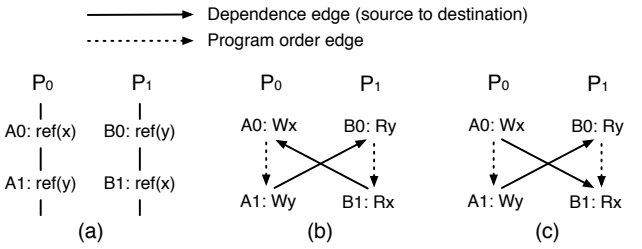

Figure 2. SC violation pattern.

However, 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 executes before B0, but A0 executes before B1. Since there is no cycle, SC is not violated. This case corresponds to the timing in Figure 1(a).

It is important to detect SCVs because, in virtually all cases, SCVs are programming mistakes — as shown in Figure 1(b), they are the result of memory-access orders that contradict a programmer's intuition. In addition, given their subtlety, they can potentially cause great harm to the program without being obvious to the programmer. Finally, the programmer cannot reproduce them using a single-stepping debugger, and has to largely rely on mental analyses of interleavings to uncover them.

Most prior work has attempted to find SCVs by focusing on detecting data races (e.g., [3, 8, 14, 15, 21, 22, 31]). However, using data races as proxies for SCVs is very imprecise. As discussed

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

above, the specific race pattern and interleaving required for an SCV is not necessarily common. In large codes, race-detection tools typically flag a very large number of data races, often causing the programmer to spend time examining races that are much less likely to cause code malfunctioning than SCVs [12, 25].

A second reason for not using data races as proxies is that we may want to uncover SCVs in codes that have intentional data races — perhaps in lock-free data structures. We may want to debug such codes 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 data races and interleavings that cause them.

Given the importance of these bugs and the difficulty in isolating them, there have been two recent proposals for hardware-supported detection of data-race cycles [20, 24]. The first one, by Lin et al. [20], focuses on detecting overlapping data races, even if they involve disjoint sets of processors. Hence, the approach is fairly conservative, resulting in false positives. However, false positives are not a problem because the goal of that approach is to avoid SCVs (by flushing the processor pipeline) rather than to detect them and report them to the programmer.

The second appoach, by Muzahid et al. [24] detects cycles that cause SCVs, precisely. However, it is only designed to work for two-processor cycles and relies on a broadcast-based cache coherence protocol in the machine. We compare our work to these two approaches in Section 8.

In this paper, we advance the state of the art by proposing the first hardware scheme that detects SCVs in a relaxed-consistency machine precisely, in a *scalable* manner, and for an *arbitrary* number of processors in the cycle. We call our scheme *Volition*. Volition 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. Volition can be used in both directory- and snoopy-based coherence protocols; it does not rely on any property of snoopy protocols such as the broadcast ability.

The current Volition design does not consider speculative loads from mispredicted branch paths. In addition, it is unconcerned with SCVs due to compiler transformations; it only reports SCVs due to hardware-initiated access reordering. Within these constraints, and with large-enough hardware structures, Volition suffers neither false positives nor false negatives for a given execution.

In our experiments, we simulate Volition in a 64-core multicore with a directory-based protocol under either the Release Consistency (RC) or the Total Store Order (TSO) model. Our results running SPLASH-2 and Parsec applications show that Volition induces negligible network traffic and execution time overhead. Moreover, by removing fences from several concurrent programs, we show that Volition can detect SCVs with several processors. Overall, Volition's preciseness, scalability and low overhead make it suitable for on-the-fly use in a variety of environments.

This paper is organized as follows: Section 2 gives a background; Section 3 shows the basic Volition mechanisms; Section 4 extends Volition to work with multiple-word cache lines; Sections 5 and 6 discuss implementation and other issues; Section 7 evaluates Volition; and Section 8 discusses related work.

2. Background

An SCV occurs when the memory accesses of a program have executed in an order that does not conform to any SC interleaving. It is virtually always a programming mistake, since it involves an unintuitive interleaving. Given its subtlety, an SCV can potentially cause great damage to the program and not be obvious to the programmer. Finally, an SCV cannot be reproduced using a single-stepping debugger, and has to be identified with mental analyses of possible interleavings.

Shasha and Snir [28] showed 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 showed the required program pattern and order of dependences at runtime for two threads. We arbitrarily assigned reads and writes to the references.

An SCV is avoided by placing one fence instruction between the two references that participate in the cycle in each thread. For example, in Figure 2, we need a fence between A0 and A1, and another between B0 and B1. The algorithm that finds where to put the fences is called the Delay Set [28].

SCVs are very subtle. A major source of SCVs is the commonly-used Double-Checked Locking (DCL) [27]. 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. Figure 3(a) shows a DCL example. The code checks variable x in access B0 and, if it is not null, it reads its field $x \rightarrow m$ in B1. If, instead, x is null, we grab a lock, check again and, if x is null, allocate a new object and assign it to x in A1. As part of the constructor, in A0, the field of the object is initialized.

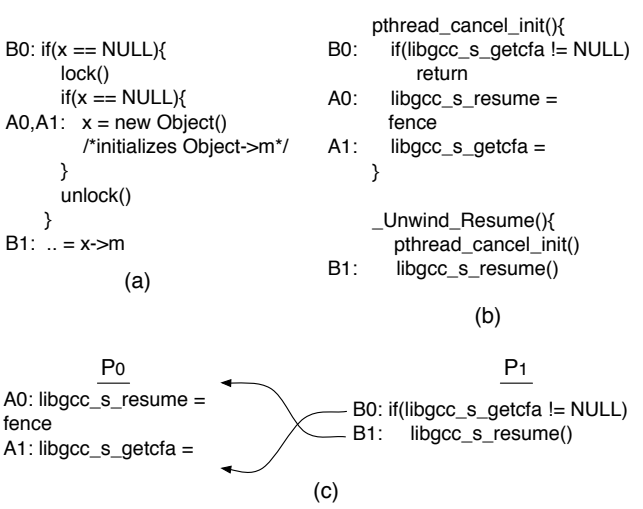

Figure 3. SCV uncovered by Muzahid et al. [24] in the Pthread library.

The DCL code has a structure like in Figure 1(a), with equivalent A0, A1, B0, and B1 references. In Figure 3(a), A0 sets the field *Object*$\rightarrow m$, and then A1 assigns *Object* to x. Unfortunately, the updates of the accesses A0 and A1 can get reordered. This causes the same problem as in Figure 1(b). According to our discussion, to guarantee SC execution, the software needs to place a fence between A0 and A1, and another between B0 and B1. Unfortunately, because the code is typically complicated, such fences end up occasionally missing.

As an example, Muzahid et al. [24] found one of such fences missing in the Pthread and Crypt libraries of glibc. The Pthread code is shown in Figure 3(b). It shows two subroutines that construct a DCL pattern. We mark the references A0, A1, B0, and B1. We see that the code has a fence between A0 and A1, but not between B0 and B1. Since the second fence is missing, an erroneous reorder with an SCV can happen. This is shown in Figure 3(c), where we picked the four relevant references. The SCV occurs when 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 put a fence between B0 and B1.

Given the importance of SCVs, there has been significant work in this area. Most prior work has attempted to find SCVs by detecting data races, which is very imprecise. There are several software-based approaches, but they typically have high overhead. Finally, there are two recent hardware-supported schemes to detect data-race cycles [20, 24]. However, they are either too conservative or are designed only for two-processor cycles and snoopy-based systems. We discuss the work in Section 8.

3. Volition: Scalable and Precise SCV Detection

In this section, we present the basic design of Volition. For now, we assume a cache line size equal to the granularity of processor accesses — e.g., one word. In Section 4, we extend the design to support multi-word cache lines.

3.1 Design Goals and Basic Assumptions

We are interested in an always-on hardware monitoring scheme usable for production runs. The scheme should detect all the SCVs that occur in the current dynamic execution, rather than attempting to find all the potential SCVs in the code. As a result, for a given binary, the scheme may detect different SCVs in different runs on the same machine and on different machines. Based on this usage model, an ideal SCV detection scheme has four traits.

1. Precise. The scheme should report SCVs, not conservative estimates of SCVs such as data races. Moreover, it should have no false positives or false negatives for a particular run.

2. Scalable. The scheme should not rely on any specific property of snoopy coherence protocols, which have limited scalability. It should be applicable to both snoopy- and directory-based protocols. In addition, the size of the hardware structures should increase only slowly with the processor count.

3. Low overhead. The scheme should have low overhead in terms of execution time and network bandwidth consumption.

4. Decoupled from the coherence protocol. Since the coherence protocol is difficult to design and verify, the SCV detection scheme should be decoupled from it.

In our design, we assume a multicore with directory-based coherence (although our scheme can also work with a snoopy-based protocol) and a relaxed memory consistency model, such as RC or TSO. We assume an MSI coherence protocol, which has a clean state (S, the line is coherent with memory and can be in multiple caches) and a dirty state (D, the line is not coherent and can be in only one cache). Each core is an out-of-order superscalar. For stores, the value is only written to the cache after retirement. Retired stores are held in the store buffer while they are being globally performed. They enter the store buffer in program order but may update the cache out-of-order. Reads can get their value before retiring. When an SCV is detected, an exception is raised.

The compiler can itself induce SCVs with certain optimizations [30]. However, Volition is a pure hardware scheme and, as a result, is not able to detect those. Hence, in this paper, we assume that the compiler does not perform SCV-inducing transformations, and we are only concerned with hardware-induced reorderings that cause SCVs. We leave the problem of preventing SCVs cooperatively by the compiler and hardware as future work.

3.2 Basic Insight to Identify an SCV

Volition detects an SCV by continuously trying to identify the pattern of Figure 2(b) across two or more processors. Such a pattern involves an interaction between processors on different memory addresses. In general, it is challenging to identify such a pattern, since the information is distributed. However, the insight of Volition is to start by first detecting a special access pattern that is a necessary condition for an SCV to subsequently occur. Such a pattern is called *Suspicious Pattern (SP)*. It can be detected locally within a processor and inexpensively. When a processor detects an SP, it piggybacks a small amount of local state information with the response coherence message. If a true SCV eventually occurs, the processors involved in the cycle will detect it. Before describing our solution, we define some terms.

Completion of a memory operation. A load completes when it gets its value from the memory system and no store from any processor can alter it; a store completes when its value updates the memory system and no load from any processor can return the value before the store. In relaxed memory consistency, memory operations from a thread can complete out of order.

Active access. A memory access *A* is *Active* when either itself or an older local access (according to program order) have not completed, *or* they are the destination of a data dependence from a remote access that is still active. For example, in Figure 4(a), B1 is active while B1 is not completed, or B0 is not completed, or (following the A1→B0 dependence) A1 is active.

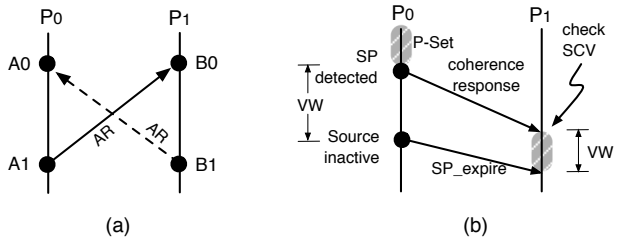

Figure 4. Basic ideas in Volition.

Pending set (*P-Set*). The *P-Set* of an access is the set of older local accesses that are still active.

Active Data Race (AR). An AR is a data race where the source access is active. We refer to the source and destination processors of an AR as AR_s and AR_d, respectively.

Suspicious Pattern (SP). A processor detects an SP when the processor is the source of an AR. The processor identifies the SP locally, when it responds to a coherence event (i.e., it provides and/or invalidates a cache line). If the local access involved in the transaction is active, then an SP is identified. For example, in Figure 4(a), assume that A1 is completed and A0 is not. Hence, A1 is active. When the dependence A1→B0 occurs, it is an AR and, therefore, P_0 detects an SP. As a result, in the response coherence message, AR_s piggy-backs some information that AR_d will need to detect an SCV, if it ever occurs. When A1 ceases to be active, then the SP is considered *expired*. At that point AR_s informs AR_d.

SCV pattern. An SCV occurs when multiple ARs form a cycle across two or more processors. Figure 4(a) shows a two-AR cycle.

Vulnerability Window (VW). Given an AR, the VW is the global physical time period during which an SCV is possible. In the AR_s processor, the VW is from the time when it identifies the SP until when the source access of the AR becomes inactive. In AR_d, the VW is from the time when it is notified about the SP (by AR_s) to when it receives the SP expiration notification by AR_s.

Volition works as follows. When an AR occurs, AR_s includes in its coherence response to AR_d some information about the *P-Set* of the AR's source access. Then, AR_d checks its local state against the information received, to flag if this AR closes a cycle. When the SP induced by this AR expires, AR_s notifies AR_d via a small *SP_expire* message so that AR_d no longer tries to check for a cycle. Figure 4(b) shows a timeline of the VWs and *SP_expire* message.

3.3 Volition Hardware Structures

To support the algorithm described, we need structures to (i) represent the local execution state, and to (ii) detect and record ARs.

3.3.1 Representing the Local Execution State

In Volition, each processor assigns a monotonically increasing *Sequence Number (SN)* to every memory access as it is issued. The SN reflects the order of local accesses in program order. If the access produces a network transaction, its SN is piggy-backed in the request message. With the proper way to handle the occasional wrap-around of SN (Section 5), the SN does not need to be very long.

The main hardware structure in Volition is the per-processor *Active Table (ACT)*. The ACT is in the core, and maintains state for all of the local accesses that are currently active. An ACT entry is allocated for each access in program order as it is issued. When an access ceases to be active and is the oldest access in the ACT, it is deallocated. At any time, the ACT may contain some entries that are completed and some that are not.

The goal of the ACT is to help record ARs. As shown in Figure 5(a), the ACT entry for an access contains its SN, the address of the location it loads or stores (*Addr*), a bit to specify whether the access is completed (*C*) and a writeback bit (*wb*). The functionality of *wb* is described later. The granularity of *Addr* (byte, half-word, word, etc.) depends on the granularity of the access. In our evaluation, we will assume word granularity only.

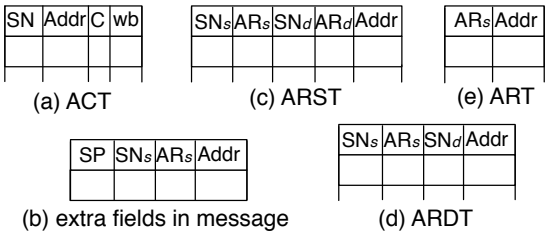

Figure 5. Hardware structures in Volition.

3.3.2 Detecting and Recording ARs

A processor detects an AR as follows. When it receives a request from the network, it checks the ACT for a race. We will later see how to avoid most of the unnecessary checks. The ACT is scanned from younger to older access, trying to find the first completed local access to the address of the request. If such an entry *E* is found, we have detected an AR. The *P-Set* is the set of accesses in the ACT that precede *E*. However, the *P-Set* is effectively encoded with the SN of *E*, which we represent as SN_s.

If an AR is found, the Volition hardware performs two actions. First, it appends the information in Figure 5(b) to the response coherence message. Such information is a bit (SP) to indicate that this is an SP, the SN of the source of the AR (SN_s), the processor ID (AR_s), and the memory address of the dependence (*Addr*).

In addition, Volition records the AR information in a local table called the *AR Source Table (ARST)*. As shown in Figure 5(c), an ARST entry contains: the AR source's SN (SN_s) and processor ID (AR_s), the AR destination's SN (SN_d) and processor ID (AR_d), and the memory address (*Addr*). AR_s obtained the values of SN_d and AR_d from the incoming message. We will explain later why we need to store AR_s: in a cycle with more than two processors, AR_s may not be the ID of the local processor.

When the destination processor of the dependence receives a coherence response with SP=1, Volition records the AR information in a local table called the *AR Destination Table (ARDT)*. As shown in Figure 5(d), an ARDT entry contains: the AR source's SN (SN_s) and processor ID (AR_s), the AR destination's SN (SN_d) and the memory address (*Addr*). We do not need to store the processor ID of the AR destination because it is the local processor.

3.3.3 Ensuring Correct Monitoring for ARs

For Volition to work correctly, a processor *P* with an ACT entry for address *Addr* has to be able to see subsequent coherence transactions to *Addr* that can cause ARs. Unfortunately, this is not guaranteed without additional support. Specifically, consider a line in state Dirty (D) in *P*'s cache that is written back to the shared cache — either (i) because it is evicted from *P*'s cache or (ii) because another processor reads it. In the first case, *P* will not be sharer in the directory anymore and, therefore, will be unable to see future reads or writes to the line; in the second case, *P* will still be a sharer in the directory, but will be unable to see future reads.

To solve this problem, when *P* writes back a D line for which it has ACT entries, Volition allocates an entry in the directory's *AR Table (ART)*. As shown in Figure 5(e), the entry contains the processor ID (AR_s) and the line address (*Addr*). In addition, *P* sets the *wb* bit in its youngest ACT entry for *Addr*.

From then on, when reads to the *Addr* by other processors reach the directory, the directory will read the ART entry and inform *P*. *P* will check its ACT and possibly send a message like the one in Figure 5(b) to the reader, informing it of an AR. Similarly, when the first write to *Addr* by another processor reaches the directory, the directory will read the ART entry and add *P* to the list of sharers that need to be notified. The ART entry will then be removed. The sharers (including *P*) may send messages like the one in Figure 5(b) to the writer if they find ARs.

When the entry in *P*'s ACT that had the *wb* bit set becomes inactive, *P* sends an *SP_expire* message to the processors it has informed of ARs, and to the directory. The latter deallocates the ART entry if it still exists.

Note that if *P* evicts a clean shared (S) line from its cache, it requires no action. The reason is that the directory is not updated, and still records *P* as a sharer.

3.3.4 Table Operations

Table 1 shows how the tables described are used. Specifically, for each of the ACT, ARST, ARDT, and ART, the table shows: (i) the condition for inserting an entry, (ii) the actions when an entry is inserted, (iii) the condition for deleting an entry, and (iv) the actions when an entry is deleted.

In the ACT, an entry is inserted when a memory instruction is issued. An entry is removed only when it satisfies the following three conditions: it is at the head of the ACT, its access is completed, and its access is not the destination of any AR. The latter means that its SN is not the SN_d of any local ARDT entry.

When an entry is deleted from the ACT, three actions need to be taken (Table 1). First, if the entry's *wb* field is set, it means that the corresponding line in L1 had been written back and, therefore, an *SP_expire* message is now to be sent to the ART to deallocate the entry. Second, the ARST is checked for entries that need to be removed; these are the entries representing ARs whose source is the removed ACT entry. Their AR_s and SN_s are equal to the local processor ID and to the SN of the removed ACT entry, respectively. Finally, Volition tries to delete the new entry at the head of the ACT, repeating the process above.

In the ARST, an entry is inserted when a new AR is detected whose source is a local access. In addition, when we discuss cycles with more than two processors (Section 3.5), we will see that we also allocate an ARST entry when an AR is *propagated* from another processor to the local one. In either case, when an ARST entry is allocated, Volition sends a message with SP=1 (format in Figure 5(b)) with information on the new AR to the AR_d processor.

Action/Condition	Active Table (ACT)	AR Source Table (ARST)	AR Destination Table (ARDT)	AR Table (ART) in dir.
Insert Condition	Memory instruction is issued (in program order)	(i) When an AR is detected locally or (ii) When a predecessor AR is propagated (cycles with >2 cores)	When a message with SP=1 is received	When a dirty cache line is written back from a cache whose processor has the line's address in its ACT
Actions on Insert		Send information on the (local or propagated) AR to the AR_d in a message with SP=1		
Delete Condition	When an entry with SN satisfies these three conditions: (i) Head of ACT (ii) Completed (C=1) (iii) SN is not the SN_d of any ARDT entry	(i) When an AR source is deleted from the ACT or (ii) When an *SP_expire* is received from a predecessor AR (cycles with >2 cores)	When an *SP_expire* is received for an AR	When directory receives: (i) either an *SP_expire* for the entry (ii) or a write request for the line in the entry
Actions on Delete	(i) If *wb*=1, then send *SP_expire* to the ART (ii) Check if any ARST entries need to be deleted (iii) Check if the ACT next top entry also needs to be deleted	Send an *SP_expire* for the (local or propagated) AR to the AR_d	Check if an ACT entry can be deleted	

Table 1. Table operations in Volition.

An ARST entry representing an AR is deleted in two cases. One is when the source reference of the AR has been removed from the ACT, as discussed above. The other is when, in environments with cycles with more than two processors (Section 3.5), an *SP_expire* message for the AR is propagated from another processor to the local one. Finally, when an entry is deleted from the ARST, an *SP_expire* message for the AR is sent to the AR_d processor.

In the ARDT, an entry is inserted when the processor receives a message with SP=1 from the source of the AR. An entry is deleted when an *SP_expire* message for the AR is received. Finally, when an ARDT entry is deleted, Volition checks if the entry at the head of the ACT can now be removed; this will be possible if the reference at the ACT head is the destination of the removed AR and of no other existing AR.

We insert an entry in the ART in the directory module when a dirty cache line is written back from a cache whose processor has the line's address in its ACT. Moreover, an entry is deleted from the ART when the directory receives (i) either an *SP_expire* message for the entry (ii) or a write request for the line in the entry.

Figure 6 repeats the information in the table for the ACT, ARST, and ARDT in the form of a state diagram. We use the diagram to describe the examples below.

3.3.5 Examples

To better understand the operations, we consider several example access streams in Figure 7. In the streams, *w* and *r* are writes and reads, and arrows are data races. Moreover, white, gray, and black circles indicate incompleted accesses, completed but active accesses, and inactive accesses, respectively.

In Figure 7(a), assume that P_1 issues r1. This causes the insertion of an entry in P_1's ACT (edge (E1) in Figure 6(a)). Since r1 reads the value produced by w1 in P_0, and w1 is active because w0 is incompleted, P_0 detects an AR. Hence, P_0 inserts an entry in its ARST ((E2) in Figure 6(a)) and responds to P_1 with a message with SP=1 ((E3) in Figure 6(a)). When P_1 receives the message with SP=1, it inserts an entry in its own ARDT ((E4) in Figure 6(a)).

Later, as shown in Figure 7(b), assume that w0 in P_0 has completed and been deleted from P_0's ACT. At this point, w1 is completed and at the head of P_0's ACT. Since w1 is not the destination of any AR (it has no associated entry in P_0's ARDT), it is deleted from P_0's ACT (edge (E5) in Figure 6(b)). Such removal causes the deletion of any entry in P_0's ARST whose source is the deleted ACT entry ((E6) in Figure 6(b)). Hence, in our example, we delete

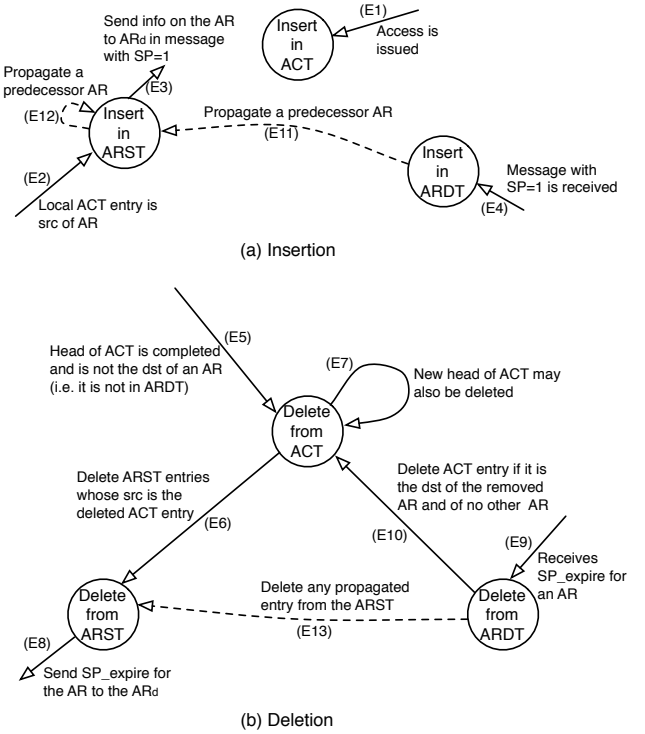

Figure 6. State diagrams for insertion and deletion of table entries. The transitions with dashed lines will be discussed later.

the ARST entry corresponding to the AR that goes from w1 to r1. After removing the ARST entry, P_0 sends an *SP_expire* message to P_1 ((E8) in Figure 6(b)). At the same time, P_0 also tries to remove its next entry in the ACT, which is w2 ((E7) in Figure 6(b)). When P_1 receives the *SP_expire*, it deletes the entry for this AR in its ARDT ((E9) in Figure 6(b)). Immediately after this, P_1 has to check its ACT — to see if the deletion of the ARDT entry makes the entry at the head of its ACT eligible for deletion ((E10) in Figure 6(b)). In the example, P_1 can remove r1's entry in the ACT.

Figure 7(c) augments 7(b) with another processor (P_2) that also had an AR whose destination is r1 in P_1. In this case, after the deletion of P_1's ARDT entry for the w1→r1 AR, P_1 cannot yet

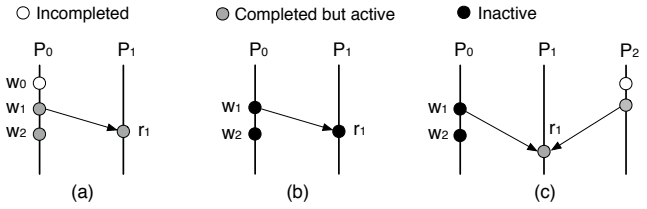

Figure 7. Examples of access streams.

remove r1's entry from the ACT. It can only be deleted when both races have ceased to be active.

3.4 Detecting SCVs between Two Processors

An SCV between two processors occurs when there are two ARs in the opposite directions forming a cycle as in Figure 8. The cycle requires that, in each of the processors, the source access of the outgoing AR is equal to or younger than the destination access of the incoming AR. Hence, the condition for an SCV is determined in a processor *locally* by comparing the local ARST and ARDT. Specifically, we are looking for an entry *arst* in ARST and an entry *ardt* in ARDT that satisfy all of the following four conditions:

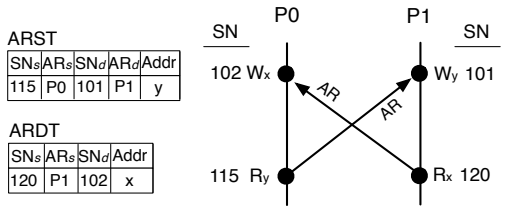

Figure 8. Detecting an SCV between two processors.

• The source processor in *arst* is the local processor. This is always the case based on our discussion so far. However, it may not be the case in cycles with more than two processors (Section 3.5).

arst[AR_s]=Local_PID

• The destination processor in *arst* is the same as the source processor in *ardt*.

arst[AR_d]=*ardt*[AR_s]

• The SN of the source access in *arst* is equal to or larger than the SN of the destination access in *ardt*.

arst[SN_s]$\geq$*ardt*[SN_d]

• The SN of the source access in *ardt* is equal to or larger than the SN of the destination access in *arst*.

ardt[SN_s]$\geq$*arst*[SN_d]

Figure 8 shows the entries in P_0's ARST and ARDT for the example shown. We see that the entries satisfy the four conditions listed above. Specifically, from top to bottom, the conditions find: P_0, P_1, 115$\geq$102, and 120$\geq$101. For simplicity, Figure 8 does not show P_1's ARST and ARDT. Using such tables, the conditions are also show to be satisfied in P_1.

In each processor, the condition for SCV is locally checked every time that a new entry is added to its ARST or to its ARDT. Specifically, when a new entry is added to the ARST, it is checked against those currently in the ARDT, and vice-versa.

With this approach, when an SCV occurs, both processors detect it. Like in Vulcan [24], the timing of the detection depends on the relative timing of the ARs. If the two writes in Figure 8 complete at approximately the same time, both processors detect the SCV when their write transaction receives the response and causes the allocation of an ARDT entry. However, if one write (say *Wy* in Figure 8) is already completed by the time its processor receives the invalidation from the other write (at the point of *Rx* in the figure), then this processor (P_1) detects the SCV as receives the invalidation. The other processor (P_0) detects the SCV as it gets the invalidation acknowledgement.

In either case, when each processor detects the SCV, it raises an exception. As in Vulcan [24], the exception may not provide the exact architectural state at the point of the SCV-causing accesses. Specifically, the information that is available to the debugger in the interrupted processor at the destination of an AR is the address being accessed, the instruction's PC and the ID of the other processor. If the destination reference of the AR 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 because the write is in the store 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 AR is the address accessed, the ID of the requesting processor, and if we augment the ACT with PCs, the instruction's PC. The exception in the source processor is not precise because newer instructions may have finished.

Execution can potentially continue after reporting the SCV in the exception handlers. It requires that Volition explicitly remove one the ARs participating in the cycle, by sending an *SP_expire* for the AR to the AR's destination processor. Otherwise, the four accesses involved in the SCV would remain active and no entry would ever be removed from the tables.

3.5 Detecting SCVs Among Any Number of Processors

In this section, we discuss the mechanism to detect SCVs involving an arbitrary number of processors. We start with some examples, describe the concept of AR propagation, and then define the conditions for an SCV.

3.5.1 Motivating Examples

Figure 9(a) shows an SCV involving three processors. The cycle is composed of active races AR0, AR1, and AR2. Although Volition can find the three ARs, our previous conditions for SCV cannot find the SCV.

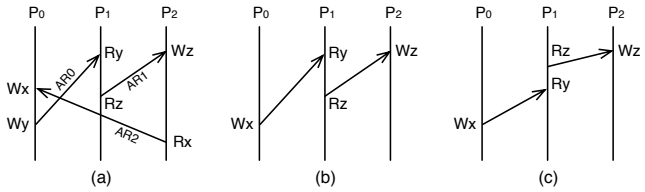

Figure 9. Finding SCVs across more than two processors.

To be able to detect the SCV, we use the insight that two ARs transitively imply another AR that combines them. For example, in Figure 9(a), ARs Wy→Ry and Rz→Wz transitively imply Wy→Wz. If P_0 and P_2 have such information, they can easily detect the SCV.

As we attempt to transitively combine ARs, we need consider the order of the local accesses of the ARs. Specifically, in a processor, the destination of one AR *has to precede* the source of the other AR. This is seen in Figure 9(b) for P_1. If the opposite is the case, as in P_1 in Figure 9(c), the two ARs cannot be combined.

3.5.2 Propagation of Active Races

To understand how Volition transitively combines two ARs, Figure 10(a) shows two ARs with the SNs of their accesses. We name these ARs from the point of view of the processor that sees them both, namely P_j: the *Predecessor* AR (AR_{pred}) is the one whose destination is in P_j, and the *Successor* AR (AR_{succ}) is the one whose source is in P_j. Such terminology does not imply the relative time of when the ARs were identified. The goal of Volition

is to generate AR_{trans}, the AR in dashes in Figure 10(b), which connects the source of the predecessor AR (SN_i) to the destination of the successor AR (SN_k).

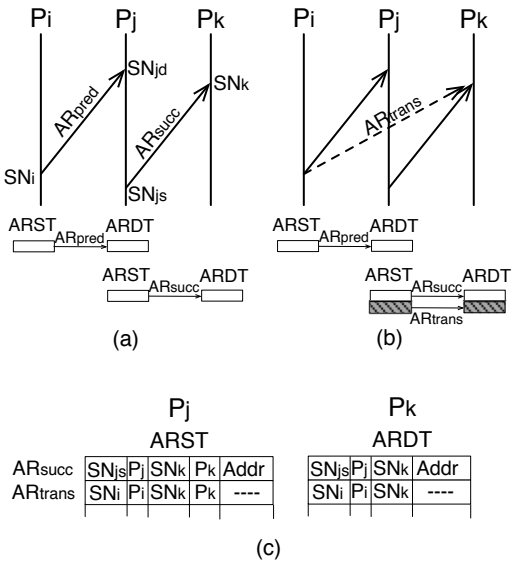

Figure 10. Propagating active races.

Figure 10(a) also shows that AR_{pred} has an entry in P_i's ARST and one in P_j's ARDT. Similarly, AR_{succ} has one in P_j's ARST and one in P_k's ARDT. Figure 10(b) shows how Volition will represent the new AR_{trans}: with a new entry in P_j's ARST and one in P_k's ARDT. They are shown as shaded. The new entries will combine information of P_i and P_k, and contain no information on P_j, even though one of the entries is in P_j.

Specifically, the new entries for AR_{trans} are shown in Figure 10(c), together with those existing for AR_{succ} for comparison. Consider the ARST for AR_{trans}. It contains source information from AR_{pred} (SN_i and P_i) and destination information for AR_{succ} (SN_k and P_k); the address field is unused. Similarly, the ARDT for AR_{trans} contains source information from AR_{pred} (SN_i and P_i) and destination information for AR_{succ} (SN_k). P_k will be unable to distinguish transitive ARs from direct ones; P_j will distinguish transitive ARs because the source information is from another processor. It appears as if Volition had "propagated" the information from P_i to P_j. Hence, we refer to the operation of transitively combining two ARs as AR *Propagation*.

3.5.3 Table Operations

To propagate ARs, the state diagrams of Figure 6 are augmented with the three transitions with dashed lines. Consider insertion first. Assume that, in Figure 10(a), P_j has already recorded AR_{pred} and now it detects AR_{succ}. After inserting the usual ARST entry, P_j observes that P_j is the destination of an AR that can be transitively combined. Hence, it creates a new entry in its ARST and sends a second message with SP=1 to the destination of AR_{succ}. This message contains the information in Figure 10(c): SN_i, P_i, and SN_k). This operation is shown in edge (E12) in Figure 6(a).

Consider, instead, that in Figure 10(a), P_j has already recorded AR_{succ} and now it detects AR_{pred}. After inserting the usual ARDT entry, P_j observes that P_j is also the source of an AR that can be transitively combined. Therefore, it performs the same operations as described above. This operation is shown in edge (E11) in Figure 6(a).

Finally, consider removal. Among AR_{pred} and AR_{succ}, the one that must become inactive first is AR_{pred}. Hence, P_i sends an *SP_expire* message to P_j. After P_j deletes its ARDT entry corresponding to (P_i, SN_i), it now has to check for an entry in its ARST with the same source (P_i, SN_i). If it finds one, it is a propagated AR. Therefore, it deletes it and sends an *SP_expire* message for it to its destination, which is P_k. This operation is shown in edge (E13) in Figure 6(b). On reception of the *SP_expire*, P_k removes its entry from ARDT.

3.5.4 SCV Condition

With the AR propagation operations, the SCV is detected when one of the created transistive ARs ends up having the same processor as source and destination. This event occurs when Volition creates a new ARDT entry in a processor, such that the AR source is also the current processor and the AR source SN is larger than the AR destination SN. Specifically, the new ARDT entry *ardt* is such that:

• The source processor in *ardt* is the local processor. Recall that there is no destination processor in *ardt* because it is always the local one.

$ardt[AR_s]$=Local_PID

• The SN of the source access in *ardt* is equal to or larger than the SN of the destination access in *ardt*.

$ardt[SN_s] \geq ardt[SN_d]$

Figure 11 shows an example with three processors. Charts (a)-(f) show snapshots of the transitive ARs as they are generated; Chart (g) shows a timeline of events. Consider Chart (a), which corresponds to time t_0. At this time, AR (1) and AR (2) have been used to generate transitive AR (i). The timeline in Chart (g) shows that, at time t_0, processor P_1 took predecessor AR (1) and successor AR (2) and generated AR (i), creating an ARST entry in P_1 and an ARDT entry in P_2.

Chart (b) shows that, at time t_1, AR (3) is detected. There are now enough ARs to creat a cycle. It will be uncovered by creating transitive ARs.

Specifically, Chart (c) corresponds to time t_2 in P_0, when P_0 generates AR (ii). The timeline in Chart (g) shows that, at time t_2, processor P_0 took AR (3) and AR (1) and generated AR (ii), creating an ARST entry in P_0 and an ARDT entry in P_1. Chart (d) is also at the same logical time t_2 in P_2, when P_2 generates AR (iv) and AR (v). As shown in Chart (g), at time t_2, processor P_2 took AR (2) and AR (3) and generated AR (iv), augmenting the ARST in P_2 and the ARDT in P_0. P_2 also took AR (i) and AR (3) and generated AR (v), recording it in the ARST in P_2 and the ARDT in P_0. This ARDT entry causes the detection of the SCV in P_0.

Potentially while this is taking place, Chart (e) shows time t_3 in P_1, where AR (iii) is created. Specifically, as shown in Chart (g), at time t_3, P_1 takes AR (ii) and AR (2) and generates AR (iii), recording it in the ARST in P_1 and the ARDT in P_2. This ARDT entry causes the detection of the SCV in P_2. Moreover, Chart (f) shows time t_4 in P_0, where (vi) is created. Specifically, as shown in Chart (g), at time t_4, processor P_0 takes AR (iv) and AR (1) and generates AR (vi), recording it in the ARST in P_0 and the ARDT in P_1. This ARDT entry causes the detection of the SCV in P_1.

When a processor detects the condition for the SCV, we envision Volition to trigger an exception. The processor then obtains the SN_s and SN_d from the offending ARDT entry. These are the local SNs of the two local accesses involved in the SCV. The processor can then read its ACT and obtain the addresses for these two accesses. Moreover, if the ACT is augmented with the program counters (PC) of the instructions, then it can also obtain the local PCs of the accesses.

From the example, we see that all of the processors involved in the cycle eventually detect the SCV. However, the actual timing and order is not deterministic. Hence, we can think of different usage modes for Volition. In one mode, as soon as the first processor detects the SCV, the processor dumps the addresses and PCs of

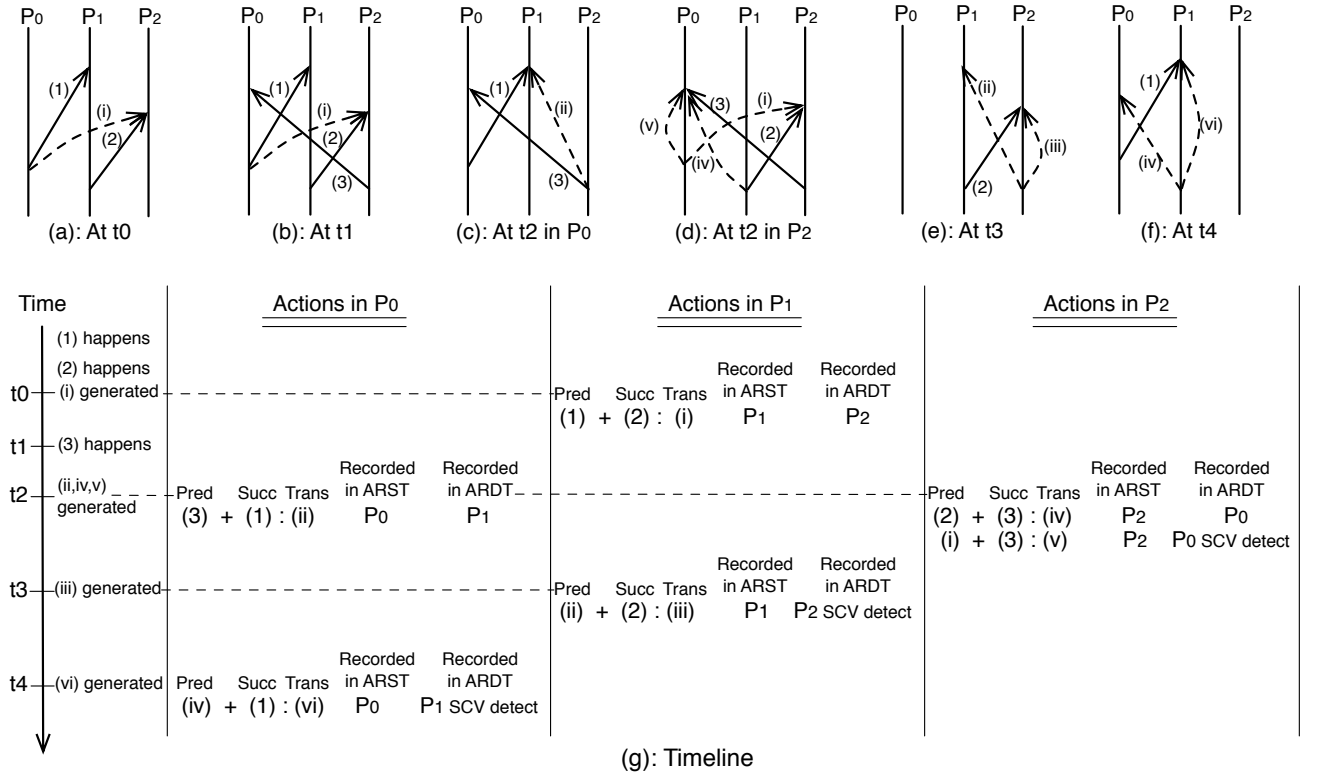

Figure 11. Detecting an SCV with three processors.

the local accesses, and then stops all other processors. In a second mode, we let each processor involved in the SCV find the SCV, dump the information, and continue. With this approach, we will get a better picture of the SCV, but the Volition tables of the processors involved in the SCV will eventually fill up and the processors will stop. Finally, in a third mode, as soon as the first processor detects the SCV, it reports it in a log file and sends an *SP_expire* for one of the ARs participating in the SCV. This will break the cycle and allow the processors to continue execution. This mode is attractive when the program runs in a non-interactive mode. The log can later be examined. However, it is not guaranteed that all the processors in the SCV will suffer an exception.

The approach described is applicable irrespectively of the number of processors participating in the SCV. In particular, it works in cycles with only two ARs, where one AR is arbitrarily chosen as predecessor and one successor. Hence, this approach supersedes the one in Section 3.4, which was presented to ease the explanation.

4. Supporting Multi-Word Cache Lines

4.1 The Problem

With single-word cache lines, every inter-processor dependence (not affected by cache displacements) induces a coherence transaction — which Volition uses for AR recording. In multi-word cache lines, the fact that all of the words in a line have to have the same state, may cause a simple design to miss some ARs (false negatives) or to falsely report some ARs (false positives). For example, in Figure 12, where *a* and *b* are in the same line, there is only one coherence action, at Wa. The Rb access is silently satisfied from the local cache. However, in reality, there are two races in the example. The opposite case, where there are no races but the protocol induces transactions, can occur due to false sharing. Hence, we must extend the Volition scheme of Section 3. In the rest of the discussion, we assume that, although the coherence protocol is line-based, coherence transactions include the address of the word accessed within the line.

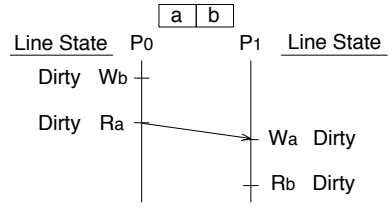

Figure 12. Missing an AR with multi-word cache lines.

4.2 Approach: Metadata Transactions

To solve this problem, we use the general approach proposed in Vulcan [24]. It involves augmenting a cache line with some information on recent accesses performed by processors to each of the words in the line. Such information should be enough to tell a processor that is referencing the line whether or not it needs to check for ARs in other processors. If it needs to, but the protocol will not generate a coherence transaction, Volition triggers a *Metadata Transaction*. Such transaction checks and updates the Volition metadata in other processors, possibly recording ARs. However, it involves no data transfer or cache coherence transition. Hence, the cache coherence protocol is unmodified.

In Volition, the information that needs to be associated with a line in a cache is the set of accesses from any processor to any of the words in the line that are currently active — i.e., that are in the Active Table (ACT) of any processor. With this information, when a processor accesses a line, it can check, for the relevant word, whether any AR can be created. In particular, we need the following information for each word:

• The most recent active write (if any).
• The set of active reads (if any) that follow the most recent active write (if there is one) or that currently exist (if there is no active write).

This information is needed when a processor issues a read or a write to the word, to indentify a potential AR. Note that all of the accesses before the most recent active write are irrelevant because they cannot source any future race.

Consequently, our basic design augments some of the cache lines with the information shown in Figure 13, which we call *Summary of Active Information* (SAI). For each word in the line, it contains: (i) the ID of the processor that has issued the most recent active write (if any), and (ii) the IDs of the processors that have issued the active reads (if any) that follow the most recent active write (if there is one) or that just currently exist (if there is no active write). Note that there is room for only a few readers. Hence, like in limited directory schemes [2], we keep space for only very few readers; if more are needed, we set a Broadcast bit. Since most of the words in a line are likely to need little or no information, the SAI should be encoded to use little space. The SAI has to travel with the line in the cache hierarchy and must be kept up to date.

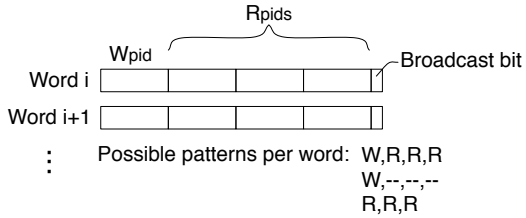

Figure 13. Structure of an SAI associated with a cache line.

When a processor accesses a line in its cache that has an SAI, Volition checks the SAI for the word accessed. If it finds that there are active accesses to this word in other processors that can create an AR with the current access, it needs to communicate with such processors. The communication automatically happens if the current access causes a coherence transaction with such processors; it simply requires including the updated SAI. Otherwise, Volition explicitly triggers a metadata transaction with the updated SAI directed to the processors that can potentially generate ARs. The arrival of the coherence or metadata transaction at these processors triggers Volition operations there.

4.3 Basic Operation

To describe the operation of Volition, we initially assume that all the cached lines have SAI entries, and that there are no cache line evictions. These assumptions will be removed later.

With these assumptions, the main challenge of the design is how to keep a line's SAI information correct. A SAI is associated with a copy of the line, and when such a copy is accessed, its SAI is updated with appropriate read or write information. If the multiprocessor cache hierarchy contained at most a single copy of a line, then keeping the SAI up to date would be easy. In reality, however, in the MSI protocol that we use in this paper (Section 3.1), a line repeatedly moves between a situation where there is a single (D or S) copy of the line in the system, and one where the caches have multiple S copies of the line that are identical.

Figure 14 shows the transition diagram for the system-wide state of a cache line in our protocol. When there is a single D or S copy of the line (leftmost circle in Figure 14), there also a single copy of the SAI. Such line may be accessed by the local processor or may receive an external write, in which case it moves to another cache, followed by its up-to-date SAI.

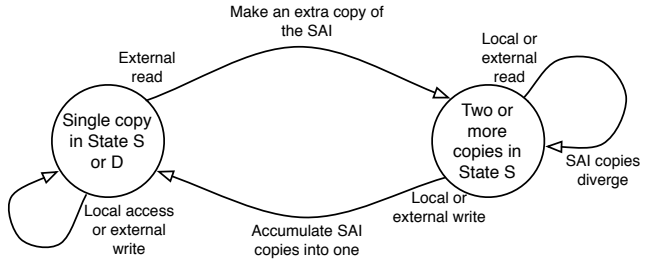

Figure 14. Transition diagram for the system-wide state of a cache line.

When an external read occurs, an extra copy of both the line and its SAI is made (rightmost circle in Figure 14). At this point, these two (or more) processors may repeatedly read words in the line. As they do so, each updates its local SAI copy, which starts to diverge from the other SAI copies. However, their divergence only consists of reader IDs, and can only induce RAW metadata transactions with the single recent-most writer of the word.

When one processor (among the sharers or otherwise) writes any of the words of the line, the state transitions back to the leftmost circle in Figure 14. As invalidation coherence messages are sent to all the sharers (which potentially record ARs), the current SAI copies are all returned to the writer. The writer accumulates the information from all of the SAI copies, and now keeps the single SAI entry for the line. The SAI information for the particular word written is reset to have a single writer ID, namely the writer processor, and no reader IDs.

We now consider several issues, namely which cache lines have SAI entries, how is the information removed from SAI entries as accesses become inactive, and the interaction with line eviction from caches.

4.4 Allocation of SAI Entries

SAI entries are only needed for lines that are being *actively* shared between processors. If a line is accessed by a single processor, it does not need an SAI. Even for a shared line, as accesses to it become inactive, Volition progressively removes information from its SAI; when the SAI contains no information, the SAI can be deallocated. Overall, at any given time, only very few lines in a multiprocessor cache hierarchy have SAI entries.

Specifically, as a processor misses on a line, if there are no sharers or the sharers do not have an SAI, then the line is read into the local cache without an SAI. Note that we also require that there is no information in the directory from a past eviction, as we will see in Section 4.6. If, instead, an SAI needs to be allocated, it is allocated in a small table in the cache controller called the Summary Active Table (SAT) (Figure 15).

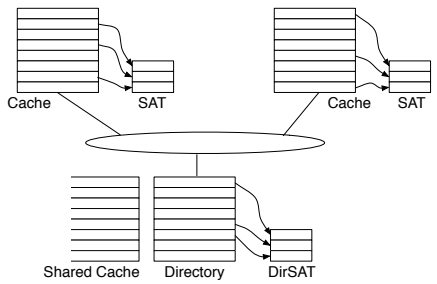

Figure 15. Storing SAI information in the system.

As a processor accesses the line, Volition attempts to update the line's SAI. However, it only does it if it finds one. As local

accesses to the line become inactive, Volition attempts to remove information from the line's SAI, if it exists. Section 4.5 discusses this operation in detail. As soon as the line's SAI becomes empty, it is deallocated from the SAT.

Finally, there are situations when a processor with a line without a SAI needs to build a SAI for it. This occurs when the processor has active accesses to the line in its ACT, and either it receives a coherence transaction for the line, or its cache wants to evict the line. In these cases, Volition allocates the SAI, updates it with information on the current local active accesses, and includes it with the coherence response or with the evicted line.

4.5 Removal of Information as Accesses Become Inactive

As an active access (by a processor *P*) to a line becomes inactive, *P*'s information should be removed from the line's SAI entries. This is because we do not want to keep querying *P* for ARs anymore in any future access by other processors.

To see how information is removed from an SAI entry, consider four possible cases. The first, trivial case is when the line is in *P*'s cache but has no SAI; in this case, no action is taken. The second case is when the line state tells us that the line is present only in *P*'s cache; in this case, Volition simply removes *P*'s ID from the SAI's write or the read area (for a write or read, respectively).

A third case occurs when the line is present in *P*'s cache and potentially in other caches as well. If *P*'s access is a read, its information is simply removed from the local SAI. It may be that other copies of the line also have the read's information, which now becomes redundant. A subsequent write will trigger an unnecessary metadata transaction to *P*'s cache. However, to miminize overall traffic, Volition takes no further action now.

However, if the now-inactive access by *P* is a write, both the local and all of the remote SAIs are updated. Volition removes the write PID from all of the SAIs to prevent future reads by other processors from initiating unnecessary metadata transactions to *P*'s cache. Hence, after Volition removes *P*'s ID from the local SAI, it initiates a metadata transaction to the directory, which is forwarded to all of the sharers of the line. All the SAI versions are updated.

Finally, the fourth case is when the line is not present in *P*'s cache — because it has been invalidated or it has been evicted. In this case, for both reads and writes, Volition initiates a metadata transaction to the directory, which is forwarded to all of the sharers of the line and removes *P*'s ID from all SAIs. This is done to avoid future unnecessary metadata transactions.

4.6 Cache Eviction of Lines with SAI Entries

When a line with SAI information is evicted from a cache, its SAI needs to be saved. The reason is that the SAI may contain unique information: the complete SAI information if this was the only cached copy of the line in the system, or unique reader PIDs otherwise. Hence, Volition stores the evicted SAI in a table in the directory controller called Directory SAT (DirSAT) (Figure 15). The DirSAT subsumes the ART structure in Section 3.3.3, which worked for single-word lines only.

Subsequent metadata transactions, as they reach the directory on their way to check for ARs in the relevant processors, they must read the dirSAT. If they find an SAI entry for the line accessed, they read its information (combining it with the information in the transaction's own SAI). Similarly, messages from processors that indicate that a certain access has become inactive, as they reach the directory, they remove the relevant bits from the SAI entry in the DirSAT.

Multiple evictions of a line's SAI from multiple caches simply accumulate their state in a single DirSAT entry. Moreover, when a line (with SAI) that is dirty in a processor is read by a second one, as the line and SAI are provided to the reader, the DirSAT also collects a copy of the SAI. The reason is that future read misses on the line will read directly from memory (since the line is not D in any cache), and they also need to obtain a copy of the SAI.

Eventually, as bits for inactive accesses are removed form an SAI entry in the DirSAT, the entry may lose all of its information and be deallocated. In addition, an SAI entry in the DirSAT is also deallocated on any write to the line. Specifically, as a write coherence transaction invalidates all copies of the line in the system and collects (and accumulates) all the SAIs for the line, it also collects and removes the SAI entry in the dirSAT.

Overall, we see that Volition manages multi-word cache lines without modifying the cache coherence protocol.

5. Implementation Issues

5.1 Wrap-Around of SNs

The wrap-around of SNs could confuse SCV detection because a number that is supposed to be comparable to another is now much smaller. While this problem occurs infrequently with our 4-Byte SN, we still need to handle it. Specifically, we use the most significant bit of the 4-Byte SN as a detector of wrap-around. When the Volition logic compares two SNs and finds that the most-significant bit of one has changed, it knows that it is a larger number that has wrapped. The range of SNs is large enough that a processor is very highly unlikely to wrap around a second time before all the other processors have wrapped around once. If this event appears to be possible, as the SN reaches a certain watermark, all processors are interrupted and the starting point of SNs is reset.

5.2 Reducing the Cost of ACT Scan

To avoid searching a processor's ACT for every incoming coherence message, we follow Vulcan's design [24] and use a counting Bloom filter (CBF) [4] to represent the current set of addresses in the ACT. If the incoming address does not hit in the CBF, we know the address is not in the ACT. We need to use a CBF because addresses need to be removed from the filter when they are deallocated from the ACT.

5.3 Region Expiration to Reduce Bandwidth

As discussed in Section 3.3, Volition causes a processor to send an *SP_expire* message when a local AR expires. While ARs are not generally very common, we can optimize this operation and save some network bandwidth when there are many clustered ARs. Specifically, rather than sending a message to the destination processor (AR_d) with the AR's SN_s immediately when the AR expires, Volition can wait for a small time. At regular intervals, when a few ARs have expired, it can multicast the *SP_expire* message with the maximum of the expired ARs' SN_s to all of the AR_ds. If ARs cluster in time, and there are only very few different AR_ds, we can save bandwidth. In practice, for our applications, we do not find this optimization beneficial.

6. Discussion

The current Volition design largely attains the design goals of Section 3.1. First, Volition detects SCVs involving an arbitrary number of processors, in a precise manner, and with no false positives or false negatives. Of course, the hardware design has to be adapted to support the finest granularity of program accesses (e.g., bytes). The current design has assumed word-level accesses.

One limitation of the current design is that it does not consider speculative loads from mispredicted branch paths. To be able to support them, we need to extend Volition, possibly delaying the recording of an AR until the source load becomes non-speculative, and discarding an AR whose destination load is proven to be in the wrong path. We consider this extension to be our future work.

Note also that Volition is not concerned with the impact of compiler optimizations 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 Volition is a dynamic scheme, it only provides information for the actual performed runs.

A second goal attained is to have a scalable design. Volition works with a scalable directory-based cache-coherence protocol. In addition, the size of its hardware structures increases only moderately with the processor count, and Volition does not need all-to-all structures.

A third goal is to have low overhead. As we show in Section 7.4, Volition incurs little execution time and bandwidth overhead.

The final goal is to be decoupled from the coherence protocol. The Volition hardware is substantial and often fairly involved. However, all of the additional messages used to manage the metadata for SCV detection are largely decoupled from the existing cache coherence protocol. Hence, the coherence protocol should not have to be revalidated.

7. Evaluation

7.1 Evaluation Setup

We focus the evaluation on three aspects, namely, (1) the ability of Volition to detect SCVs, (2) the characteristics of SCVs, and (3) the overhead and scalability of Volition.

We implement Volition in the SESC [26] cycle-level architectural simulator. We model a multicore with 64 cores and an MSI directory-based cache coherence protocol. The hardware uses either the Release Consistency (RC) or the Total-Store-Order (TSO) memory consistency model. We use different store buffer sizes to see their impact on access reordering. For comparison, we also implement the Conflict Ordering scheme of Lin et al. [20]. We name it *CO*. Such scheme, as we describe in Section 8, enforces SC. When a potential SCV is about to occur, CO avoids it by squashing and replaying certain instructions. The configuration of the simulated machine is shown in Table 3.

Architecture	Multicore chip with 64 cores
Core width; ROB size	4-issue; 128 entries
Consistency	RC or TSO
Store buffer	32 entries
Private L1 cache	32KB WB, 4-way, 2-cycle round trip
Shared L2 cache	1MB module/proc. Module: WB, 8-way
Latency to L2	Local module: 11-cycle round trip
Cache line size	32 bytes
Cache coherence	Directory-based MSI protocol
Network	2-D mesh with 7-cycle hop latency
Main memory	200-cycle round trip
Volition parameters	SN size: 4 bytes; ACT size: 256 entries per-node ARST, ARDT: 40 entries each; per-node SAT: 100 entries

Table 3. Architecture parameters.

We run the applications shown in Table 4. They include several small codes with concurrent algorithms, a kernel with a double-checked lock (DCL), and SPLASH-2 and Parsec applications. The codes with concurrent algorithms were mostly obtained from [1], which in turn comes from CheckFence [5]. They are small C-code programs where threads share data structures without synchronization, and rely on explicit fences to maintain correct access ordering. Many of these programs insert and remove elements from a linked list. In addition, we added two well-known algorithms for mutual exclusion, namely Dekker (which only runs with two threads) and Peterson.

We use the codes with the concurrent algorithms and DCL to check the ability of Volition to detect SCVs. Specifically, we explicitly remove all of their fences and run each code 100 times. We count the total number of SCVs detected. Note that many of the executions are now incorrect, but they help us understand Volition's effectiveness. Finally, we use the SPLASH-2 and Parsec codes to evaluate the overheads and scalability of Volition.

Set	Application	Description
Conc. Algo.	Aharr	Variant of Harris
	Dekker	Algorithm for 2 proc. mutual exclusion
	Harris	Non-blocking set
	Lazylist	List-based concurrent set
	Moirbt	Non-blocking sync. primitives
	Moircas	Non-blocking sync. primitives
	Ms2	Two-lock queue
	Msn	Non-blocking queue
	Mst	Non-blocking queue
	Peterson	Algorithm for N proc. mutual exclusion
	Snark	Non-blocking double-ended queue
Bug	DCL	Double-checked lock without fence
Full Apps	SPLASH-2	12 programs
	Parsec	4 programs

Table 4. Applications executed.

7.2 Ability to Detect SCVs

To assess Volition's ability to detect SCVs, we take each of the small codes stripped of fences and run them 100 times. We use the simulator to count the number of SCVs observed with Volition — even if the SCVs repeat across runs. When a processor dectects an SCV, it sends an *SP_expire* to remove one of the ARs and ensure that the program continues execution. We model either RC or TSO.

Table 2 shows, for each consistency model, the number of SCVs detected with full Volition support (*# of SCVs*). It also shows the SCVs detected when Volition does not keep per-word access information (*# of Line-SCVs*). This environment is missing the SAT support from Section 4 that records which words of the line have been accessed by the processor. In this case, there are no metadata transactions — only the transactions generated by the coherence protocol occur. Hence, Volition misses SCVs. Finally, the table shows the number of active races detected (*# of ARs*). Intuitively, these are the races between largely concurrent accesses. They roughly capture the types of races recorded by DRFx [22] and Conflict Exceptions [21].

Looking at the RC columns, we see that Volition detects many SCVs in these codes. If metadata transactions are disabled (*Line-SCVs*), about half of the SCVs are missed. Hence, we need full Volition support. We also see that the number of ARs is very high — on average, over two orders of magnitude higher than the number of SCVs. This shows that ARs are not good proxies for SCVs.

The number of SCVs detected changes with the memory model. Typically, the more relaxed RC model causes more SCVs and Line-SCVs than the TSO model. However, in some applications, the opposite is the case.

Finally, the table shows the number of replays required to enforce SC in CO from Lin et al. [20], running on RC. We can see that the number of replays is lower than the number of ARs. However, it is, on average, one order of magnitude higher than the number of SCVs in Volition. Therefore, we conclude that, while CO is more precise than just reporting active races, it is much less precise than Volition in its detection of SCVs.

7.3 Characteristics of SCVs

In this section, we characterize the SCVs observed. Figure 17 takes the SCVs reported for each code and memory model in Table 2 and classifies them based on the number of processors participating in

Appl.	# of Runs	RC			TSO			CO
		# SCVs	# Line-SCVs	# ARs	# SCVs	# Line-SCVs	# ARs	# Replays
Aharr	100	166	20	4097	29	29	7035	2443
DCL	100	122	48	8529	61	55	9570	3984
Dekker	100	316	140	2653	675	235	7758	763
Harris	100	123	56	11752	43	39	9023	4061
Lazylist	100	49	26	11688	85	76	11535	2402
Moirbt	100	127	37	9408	92	61	12618	2824
Moircas	100	90	41	9522	14	14	12467	2969
Ms2	100	98	93	26252	256	176	26563	5891
Msn	100	355	251	15277	393	204	22717	1689
Mst	100	2288	1048	110293	875	618	181644	8589
Peterson	100	31	23	2028	40	24	2860	615
Snark	100	174	103	213082	276	60	196909	16432
Average	100	328	157	35381	236	132	41724	4388

Table 2. Volition's ability to detect SCVs.

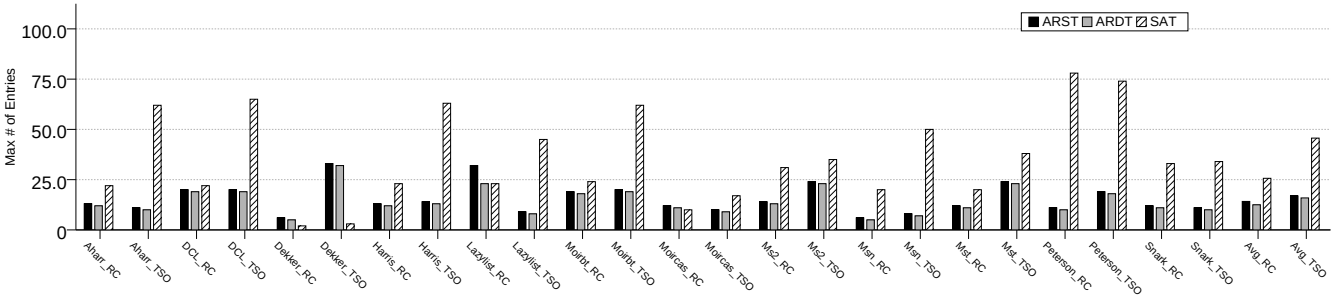

Figure 16. Table size requirements.

the SCV cycle. Specifically, we have cycles with 2 processors, 3 processors, and 4 or more processors. For each code, the number of SCVs is normalized to the number under RC.

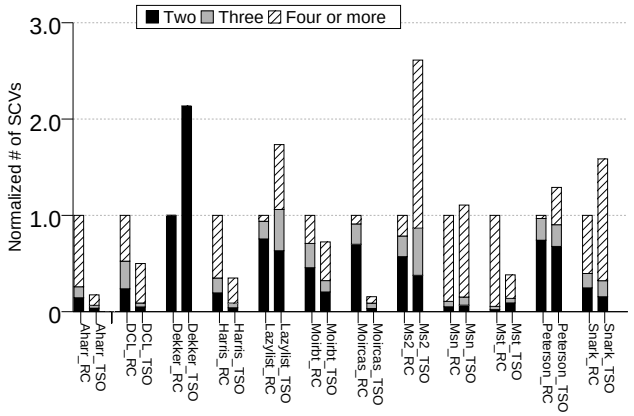

Figure 17. Number of processors involved in an SCV.

The figure shows that, in these sharing-intensive, fence-free codes, cycles appear with a variety of processor counts. While two-processor SCVs dominate in some applications (e.g., Moircas for RC), four-and-more processor SCVs are dominant in others (e.g., Msn). Hence, Volition's ability to detect cycles with an arbitrary number of processors is useful for the bug conditions represented by these fence-free codes. Note that Dekker can only have 2-processor cycles.

Figure 16 shows the use of the Volition hardware tables. Specifically, it shows, for each program and memory model, the maximum number of entries in use in each of the per-node ARST, ARDT, and SAT. We can see that these sizes are modest. In most cases, the maximum number of entries used in the ARST and ARDT is less than 25. For the SAT, it is less than 75. Therefore, our proposed sizes of 40 entries for the ARST and ARDT, and 100 entries for the SAT (Table 3) are more than enough.

Figure 18 shows the sensitivity of the number of SCVs to the size of the store buffer. For each program under RC, the figure shows the number of SCVs with store buffers of 4, 8, 16, and 32 entries. The latter is the default size. For each program, the bars are normalized to the number of SCVs for 4-entry buffers and are broken down into the number of processors per cycle.

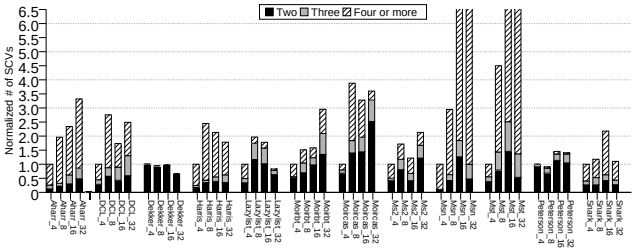

Figure 18. Sensitivity to the size of the store buffer.

Intuitively, larger store buffers should induce more SCVs because they allow more store reordering. While this is the general trend, there are several programs where smaller buffers induce more SCVs. The reason is that smaller buffers also affect the timing

of execution significantly, by introducing more access stalls due to full buffers.

7.4 Overheads of Volition

We consider two overheads of Volition, namely the increase in network traffic and the increase in program execution time. Figure 19 shows the total number of bytes transferred in the network of the machine for different programs and memory consistency models. The figure also includes bars for the average of the SPLASH-2 applications and the average of the Parsec codes. We break down the bytes transferred into those from memory access requests (*MemAcc*), data transferred in a read (*Read*) or write (*Write*), coherence activity in invalidations, acknowledgements, or forwarding to the owner cache (*Coh*), and additional traffic due to Volition (*Overhead*). The latter includes messages such as *SP_expire*, AR propagation in cycles with more than two processors, or SAI information transfer. The sum of the first four categories is normalized to 100. From the figure, we see that, even with 64 processors, the additional traffic induced by Volition (*Overhead*) is largely negligible. For fewer processor counts, it is even smaller.

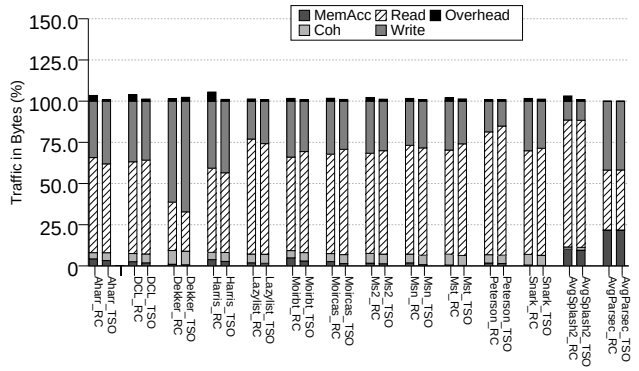

Figure 19. Network traffic overhead of Volition.

Finally, we consider the execution time overhead of Volition in Figure 20. The figure shows the execution time of the different applications under the RC memory model on a multiprocessor without Volition (Baseline) and on one with Volition. We show bars for all the small programs, their average, the average of SPLASH-2, and the average of Parsec. For each application, the bars are normalized to the Baseline.

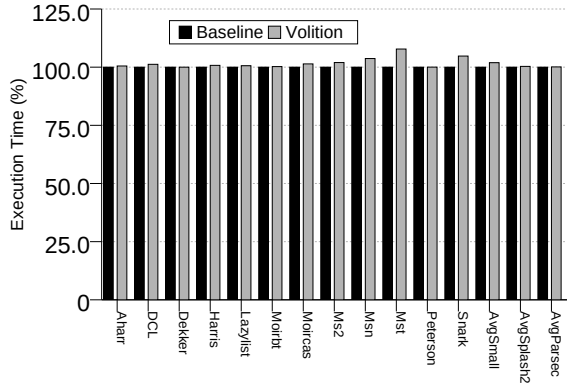

Figure 20. Execution overhead of Volition.

The main source of execution overhead in Volition is the stall due to a full ACT. We use a 256-entry ACT, and we can see that, for most applications, the execution time overhead of Volition for 64 processors is negligible. Some of the applications with visible overhead are those with many SCVs which, according to Table 2 include Mst and Msn. Still, for the large applications (SPLASH-2 and Parsec), there is no visible overhead. For the small applications, the average overhead is only about 2%.

Overall, based on the results of the traffic and execution time overheads, we conclude that Volition has low overhead and good scalability.

8. Related Work

We discuss related work in architecture, compilation, testing, and hardware verification. In architecture, the most related work is Vulcan by Muzahid *et al.* [24] and Conflict Ordering (CO) by Lin *et al.* [20]. Both works are based on identifying SCVs in hardware using Shasha and Snir [28] delay sets.

Vulcan [24] is the most similar work to Volition. It is a hardware scheme to detect SCVs at runtime, in programs running on a relaxed-consistency machine. It also leverages the cache coherence transactions to detect dependence cycles between processors and, from there, SCVs. It also supports multiple-word cache lines. While it has similar metadata structures as Volition, it works differently. It relies on a snoopy-based coherence protocol. Moreover, the design presented only operates with 2-processor SCVs. With Volition, we have taken a different approach, focusing on scalability and on handling SCV cycles with an arbitrary number of processors. The resulting Volition design is scalable, as it works with a scalable directory-based cache-coherence protocol and its hardware does not need all-to-all structures. In addition, it works seamlessly for any number of processors in the SCV cycle.

CO [20] is a technique that detects upcoming SCVs and enforces SC in a relaxed-consistency machine. As a potential SCV is about to occur, CO avoids it by squashing and replaying certain instructions. Although it is an SC enforcement scheme, it can be used as an SCV detection scheme if it reports the SCV when the replay is needed to retain SC semantics. However, CO has substantial false positives, which are fine in an SC enforcement approach but not in an SCV detection scenario. To see why, consider Figure 21. Initially, there is a race between P_0 and P_1 on variable x. When CO sees this, it gets from the directory the set of pending writes. In this example, it gets the writes to variables y and z. If P_1 then tries to access y or z, CO conservatively assumes an SCV is about to occur, and causes a replay. Clearly, these dependences do not cause a cycle; we need a new dependence between P_1 and P_0 for a cycle.

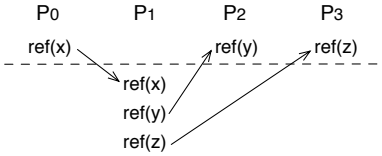

Figure 21. Operation of CO.

CO also requires the serialization of some of the accesses from the same processor to operate correctly. Finally, it is unclear how CO works for a distributed directory design: since a processor gets the pending sets asynchronously from the directory modules, the information seen by different processors can easily become inconsistent.

Other work has focused on identifying data races as proxies for SCVs. However, data races and SCVs are very different, and programs have more data races than SCVs. Specifically, one line of

work detects incoming coherence messages on data that has local outstanding loads or stores. This work includes that of Gharachorloo and Gibbons [14] and many aggressive speculative designs (e.g., [3, 8, 15, 31]). Another line of work detects a conflict between two concurrent synchronization-free regions. This includes DRFx [22] and Conflict Exceptions [21]. 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.

There are compiler techniques to identify race pairs that could cause SCVs, typically using the Delay Set algorithm, and then insert fences to prevent cycles (e.g., [13, 16, 18, 29]). They are conservative because they only use static information, and typically cause large slowdowns. Lin *et al.* [19] can hide some of the resulting fence delay with architectural support. Duan *et al.* [11] use a race detector to construct a graph of races dynamically. Then, off-line, they traverse the graph to find potential SCVs. Our work 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., [5–7]). 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., [9, 10, 23]). 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. Conclusions

This paper proposed Volition, the first hardware scheme that detects SCVs in a relaxed-consistency machine precisely, in a scalable manner, and for an arbitrary number of processors in the cycle. Volition uses cache coherence protocol transactions to detect cycles in memory-access orders across threads. When a cycle is about to occur, an exception is triggered. For the conditions considered in this paper, Volition suffers neither false positives nor false negatives. We simulated Volition on a 64-processor multicore with directory-based coherence and 32-byte cache lines, running some small codes and SPLASH-2 and Parsec applications. Our results showed that Volition induces negligible network traffic and execution time overhead and is scalable. In addition, it detects SCV cycles with several processors. Volition is suitable for on-the-fly use.

Acknowledgements

This work was supported in part by NSF grants CCF-1012759 and CNS-1116237; Intel under the Illinois-Intel Parallelism Center (I2PC); Spanish Gov. & European ERDF under grants TIN2010-21291-C02-01 and Consolider CSD2007-00050; and NSF China grants 61073011 and 61133004.

References

[1] CheckFence at SourceForge. `http://sourceforge.net/projects/checkfence/`.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *International Symposium on Computer Architecture*, May 1988.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Int. Symp. on Computer Architecture*, 2009.

[4] F. Bonomi et al. An Improved Construction for Counting Bloom Filters. In *Ann. Euro. Symp. on Algo.*, Sep 2006.

[5] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *Programming Language Design and Implementation*, Jun 2007.

[6] S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *Computer Aided Verification*, Jul 2008.

[7] 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.

[8] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[9] Y. Chen, L. Yi, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan. Fast Complete Memory Consistency Verification. In *International Symposium on High Performance Computer Architecture*, Feb 2009.

[10] A. Deorio, I. Wagner, and V. Bertacco. DACOTA: Post-silicon Validation of the Memory Subsystem in Multi-core Designs. In *International Symposium on High Performance Computer Architecture*, Feb 2009.

[11] 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 *Code Gen. and Opt.*, Mar 2009.

[12] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-race Detection for the Kernel. In *Operating System Design and Implementation*, Feb 2010.

[13] X. Fang, J. Lee, and S. P. Midkiff. Automatic Fence Insertion for Shared Memory Multiprocessing. In *International Conference on SuperComputing*, Jun 2003.

[14] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *Symp. on Par. Alg. and Arch.*, Jul 1991.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, May 1999.

[16] A. Krishnamurthy and K. Yelick. Analyses and Optimizations for Shared Address Space Programs. *Jour. Paral. Dist. Comp.*, Nov 1996.

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

[18] J. Lee and D. A. Padua. Hiding Relaxed Memory Consistency with a Compiler. *IEEE Trans. Comput.*, Aug 2001.

[19] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency using Conditional Fences. In *Parallel Architecture and Compilation Techniques*, Sep 2010.

[20] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *Architectural Support for Programming Languages and Operating Systems*, Mar 2012.

[21] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-races. In *International Symposium on Computer Architecture*, Jun 2010.

[22] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *Programming Language Design and Implementation*, June 2010.

[23] A. Meixner and D. J. Sorin. Dynamic Verification of Sequential Consistency. In *Int. Symp. on Comp. Arch.*, Jun 2005.

[24] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *International Symposium on Microarchitecture*, December 2012.

[25] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races using Replay Analysis. In *Prog. Lang. Des. and Impl.*, Jun 2007.

[26] 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.

[27] 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. Des. Conf.*, 1996.

[28] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[29] 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 *Prin. and Pract. of Para. Prog.*, Jun 2005.

[30] J. Ševčík. Safe Optimisations for Shared-memory Concurrent Programs. In *Prog. Lang. Des. and Impl.*, Jun 2011.

[31] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.