SCsafe: Logging Sequential Consistency Violations Continuously and Precisely *

Yuelu Duan, David Koufaty,† and Josep Torrellas

University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

†Intel Labs
david.a.koufaty@intel.com

ABSTRACT

Sequential Consistency Violations (SCV) in relaxed consistency machines cause programs to malfunction and are hard to debug. While there are proposals for detecting and recording SCVs, they are limited in that they end program execution after detecting the first SCV because the program is now non-SC. Therefore, they cannot be used in production runs. In addition, such proposals rely on complicated hardware.

To address these problems, this paper proposes the first architecture that detects and logs SCVs in a continuous manner, while retaining SC. In addition, the scheme is precise and uses substantially simpler hardware. The scheme, called *SCsafe*, operates continously because, after SCV detection and logging, it recovers and resumes execution while retaining SC. As a result, it can be used in production runs. In addition, SCsafe is precise in that it identifies only true SCVs — rather than dependence cycles due to false sharing. Finally, SCsafe's hardware is mostly local to each processor, and uses known recovery techniques. We evaluate SCsafe using simulations of 16-processor multicores with Total Store Order or Release Consistency. In codes with SCVs, SCsafe detects and reports SCVs while enforcing SC during the execution. In codes with few SCVs, it adds a negligible performance overhead. Finally, SCsafe is scalable with the processor count.

1. INTRODUCTION

Programmers writing and debugging shared-memory programs assume Sequential Consistency (SC). Under SC, the memory operations of the program must appear to execute in some global sequence as if the threads were multiplexed on a uniprocessor [15]. In practice, memory accesses are pipelined, overlapped, and reordered by the hardware. Unless the program uses correct synchronization to prevent unwanted reorders, an SC violation (SCV) may occur, which is very hard to debug.

As an example, consider Figure 1(a). Processor P_1 initializes variable a and then sets flag OK; later, P_2 tests the flag and, if set, uses a. If the hardware reorders the writes of P_1 as shown with the arrows, where the initialization of a is delayed, P_2 ends up reading the uninitialized a. This order is an SCV.

An SCV occurs when there is a dependence cycle [28]. For a two-threaded SCV, two conditions need to be true. First, we need to have two data races. In the example, we have races on variables a and OK. Second, at runtime, these races must overlap in time and intertwine in a manner that forms a cycle. This is shown in Figure 1(b). The dashed arrows show program order, while the solid ones show the order of dependences: A_2 executed before A_3 (arrow (1)), while A_4 executed before A_1 (arrow (2)), forming a cycle. A cycle can be formed with any number of threads.

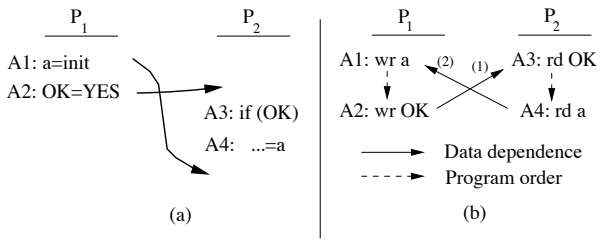

Figure 1: Example of an SC Violation (SCV).

An SCV is a type of concurrency bug that, while not as common as popular bugs like data races, is important for three reasons. First, it can induce serious harm by causing a program to execute totally counter-intuitively. Second, it is hard to debug, as it depends on the timing of events, and single-stepping debuggers cannot reproduce it. Finally, it is often concentrated in critical codes, such as those that perform fine-grain communication and synchronization — synchronization libraries, task schedulers, and run-time systems.

There are proposals of hardware schemes to detect and record SCVs [19, 20, 21, 24]. However, they have limitations. Specifically, some schemes are very conservative, as they look for only a single data race where the two participating accesses are concurrent [19, 20].

The other schemes look for dependence cycles [21, 24]. However, they terminate execution after detecting the first SCV. This is because the program state is now non-SC and, therefore, incorrect. Further execution could find artificial additional SCVs. This approach is incompatible with production runs and, therefore, suboptimal, as some SCVs may happen only during production runs. Instead, we would like to log the SCV bug for later debugging, and continue at production-run speeds under strict SC-enforced execution, in order to correctly capture future SCVs. A second limitation of these schemes is that they rely on complicated hardware.

To solve these limitations, this paper proposes the first architecture that detects and logs SCVs in a *continuous* manner, while retaining SC. The scheme is called *SCsafe*. In SCsafe, when a processor P_i executes an out-of-order access A, the hardware in P_i prevents other processors from observing it by rejecting coherence transactions received by P_i directed to the address accessed by A. P_i only responds when all local accesses prior to A finish. When two or more pro-

* This work was supported in part by NSF under grants CCF-1012759 and CCF-1536795. Yuelu Duan is now with Whova.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

cessors reject each other's requests and cause a deadlock, a dependence cycle (and, hence, an SCV) has just been prevented from happening. In this case, SCsafe quickly detects the deadlock, records the SCV, and recovers and resumes execution while maintaining SC. As a result, SCsafe operates under SC continuously, and can be used in production runs.

SCsafe is precise in that it records only true SCVs — rather than dependence cycles due to false sharing. Also, its hardware is simpler than prior schemes because it is mostly local to each processor, and uses known recovery techniques.

SCsafe is a pure hardware scheme and, therefore, considers only access reordering *induced by the hardware*. The compiler could itself induce SCVs with certain optimizations [27], but this is outside of SCsafe's scope. It requires passing information between compiler and hardware.

We evaluate SCsafe using simulations of 16-processor multicores with Total Store Order (TSO) or Release Consistency (RC). The results show that SCsafe is effective. In codes with SCVs, SCsafe detects and reports SCVs, while enforcing SC during the execution. In codes with few SCVs, it adds a negligible performance overhead. Finally, SCsafe is scalable with the processor count.

This paper is organized as follows: Section 2 gives a background; Section 3 presents the idea in SCsafe; Sections 4–5 introduce the different parts of SCsafe; Sections 6–7 evaluate it; and Section 8 covers related work.

2. BACKGROUND

2.1 Definitions

For a memory instruction, this paper uses the terms *performed*, *retired*, *finished*, and *M-speculative*. A load has *performed* when the processor receives the loaded data from the memory system. It *retires* when it reaches the head of the Reorder Buffer (ROB) and has performed. After retirement, the load is *finished*.

A store *retires* when it reaches the head of the ROB and its address and data are available. The store is deposited into the write buffer. After this, when the memory consistency model allows, the store is merged with the memory system, potentially triggering a coherence transaction. When the coherence transaction terminates (e.g., when all the invalidation acknowledgments have been received), the store has *performed*, and is now *finished*.

The memory consistency model supported by the hardware determines the access reorders that are legal. In TSO [1], a load can perform before earlier (in program order) stores but not before earlier loads; a store cannot perform before earlier accesses. In RC [12], a load can perform before earlier accesses; a store can also perform before earlier accesses but, to keep precise exceptions, such earlier accesses are in practice restricted to other retired stores. This is what we assume in this paper.

Hardware implementations typically allow loads to perform earlier than allowed by the memory consistency model — as long as the load is not observed by other processors [11]. A local load is observed when the processor receives a coherence transaction directed to the address read by the load. Consider the time between when a load is performed and when it is allowed to be performed according to the memory consistency model. During this time, we say that the load is *M-speculative* (or *speculative relative to the memory consistency model*). We use this term to mean that its status depends on the consistency model supported by the system. For example, consider a load (l_2) that performs while an earlier one in the pipeline (l_1) is not yet performed. Under RC, l_2 is not M-speculative; under TSO, l_2 is.

While a load is *M-speculative*, if it is observed, the load and subsequent instructions are squashed. When the load ceases to be M-speculative, if it is observed, it will not be squashed.

2.2 Detecting SCVs and Enforcing SC

An SCV occurs when threads participate in a cycle of data dependences and program orders [28] (Figure 1(b)). There are several hardware schemes for SCV detection [19, 20, 21, 24]. They can be classified into conservative and highly specific. The former [19, 20] look for a fairly conservative necessary condition for SCV: a data race where the two participating accesses are concurrent. This is very conservative because most such races are not accompanied by a second, cycle-forming race.

The highly-specific schemes (i.e., Vulcan [21] and Volition [24]) leverage cache-coherence transactions to dynamically track the data dependences between processors, looking for a cycle pattern like Figure 1(b). While they are effective at finding SCVs, they have two limitations.

The first one is that, after they find the first SCV in a program, they are unable to retain SC. The program state is now non-SC and, therefore, incorrect. As a result, they terminate execution, perhaps with a crash. Hence, these schemes are incompatible with production runs and, therefore, suboptimal, as some SCVs may only happen during production runs.

The second limitation is that they use complex hardware. They introduce elaborate hardware structures for metadata. They time-stamp the dynamic accesses of processors, and then compare the time-stamps when processors communicate — all in hardware. The time-stamps are passed in augmented or special coherence transactions. Word-level dependence disambiguation is attained with additional per-word state and especial transitions (Vulcan) or with special hardware structures and new cache coherence transactions (Volition). In SCsafe, we use simpler, mostly-local hardware.

A related approach is to use hardware to only *enforce SC* (e.g., [4, 7, 11, 13, 18, 25, 33]). These schemes look for a necessary condition for an SCV and, when detected, squash instructions to force the threads away from the SCV path. In most schemes, the necessary condition is the presence of an access that is observed while it is M-speculative relative to SC. The access can be a load or, with support for speculative caches, a store. In Figure 1(b), the access is store A_2.

While these schemes are useful for their purpose, they are not usable to detect SCVs. This is because, when they squash instructions to avoid the SCV, they discard the state that would be needed for SCV detection and recording. In addition, in most cases, as we will see, we would have a false positive because no SCV would end up happening. We describe these schemes in Section 8.

2.3 Why Continuous Detection of SCVs?

Finding SCVs is important for several reasons. First, SCVs

cause a program to execute in totally counter-intuitive manners. Second, there are no software techniques for SCV detection and recording. Third, SCVs are very hard to debug, as single-stepping debuggers cannot reproduce them.

However, one can ask: (i) why not simply look for data races, (ii) why not limit the design to SC enforcement only, or (iii) why require the SCV detection to be continuous?

We target SCVs and not data races because SCVs are much more harmful than most data races. In commercial codes, race-detection tools find *many* races. Typically, a busy developer does not consider many of them important enough to devote her attention to them [9, 22]. Instead, she prefers to focus on those most likely to cause malfunction. Among these are SCVs, which require two or more overlapping data races in a cycle. Only a very small fraction of data races are associated with an SCV [21]. 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 some types of lock-free data structures.

SCVs would disappear from high-level language code if programmers annotated all racing accesses as volatile (in Java) or atomic (in C++). However, programmers often fail to do so, possibly involuntarily. There is substantial existing code without these annotations.

Only enforcing SC rather than also recording SCVs is not enough. The developer needs to know about SCVs that were avoided, and debug them later, for two reasons. First, a latent SCV is a sign of a deep bug; such bug may have other ramifications beyond causing SCVs, like changing code state. Second, we would like the program to also run on off-the-shelf machines correctly.

Finally, providing continuous SCV detection is important because SCVs are timing-dependent and unpredictable. Hence, they need to be caught at production-run speeds, possibly during production runs. During a production run, terminating or crashing the program at the first SCV is unacceptable.

3. IDEA: CONTINUOUS & PRECISE SCV DETECTION

SCsafe is the first hardware architecture for relaxed consistency machines that detects and logs SCVs in a continuous manner. This makes it different from past proposals. With SCsafe, as a program executes at production run speeds, the hardware records any SCVs that occur (for later debugging) while ensuring that the execution is always SC. In addition, SCsafe is precise (i.e., has no false alarms due to false sharing) and has modest hardware cost.

A processor's SCsafe hardware dynamically keeps track of all the out-of-order accesses that are not M-speculative relative to the consistency model of the machine (and hence would not be squashed if observed). Then, it stalls any incoming coherence transaction directed to any of these out-of-order accesses. When two or more processors that reject each other's requests cause a deadlock, a dependence cycle (and, hence, an SCV) has just been prevented from happening. At this point, SCsafe's hardware automatically detects the deadlock and logs the SCV — i.e., the deadlocked instructions' program counters and addresses accessed.

SCsafe then forces at least one of the threads involved in the deadlock to roll back the out-of-order accesses and re-execute them. During this process, SC is retained. As execution continues at production-run speed, the machine is able to detect and record any future SCVs that occur. These will be true SCVs, not "artificial" ones that could have been "fabricated" if SC had not been enforced during the whole process.

As we will see, the SCsafe hardware is relatively simple: it is mostly local to each processor, and uses known mechanisms for state recovery. Moreover, it is scalable. The key to hardware simplicity over Vulcan and Volition is to never satisfy a request that may end-up closing a dependence cycle, but stall it. Then, we do not need timestamps to identify an SCV: we simply look for a deadlock cycle. Fortunately, then, no incorrect data has been supplied, and correct execution can resume.

4. SCsafe OPERATION

4.1 Reordered Accesses and SCVs

To understand SCsafe, we first define the concept of *Reordered accesses*. Intuitively, these are performed accesses that follow (in program order) unfinished accesses from the same processor, but are allowed by the memory consistency model to be visible to other processors. Other processors can read and write the data accessed by the Reordered accesses without squashing the Reordered access instructions.

Formally, a *Reordered access* in a thread is a load or a store instruction for which all of the following is true:

• Has performed — i.e., for a load, it has brought the data from the memory system and, for a store, the coherence transaction that it triggered has completed.

• It follows in program order at least one unfinished memory access in the same thread: an earlier load has not yet performed and retired, or an earlier store has not yet retired and performed.

• It is not M-speculative. Hence, if the processor receives an external coherence transaction for the data that the instruction accessed, the instruction is not squashed.

Different memory consistency models allow different types of Reordered accesses. In TSO, given an unfinished store, all the loads that follow it in program order, up to (but not including) the earliest not-yet-performed load are Reordered. In TSO, an unfinished load cannot have any Reordered accesses.

In RC, given an unfinished store, all the performed loads and performed stores that follow it in program order are Reordered. Given an unfinished load, all the performed loads that follow it in program order are Reordered. (There cannot be any performed store that follows an unfinished load).

An SCV occurs when two or more processors form a dependence cycle. A necessary condition for a cycle is that a processor (P_1) has a Reordered access (A_1) that: (i) conflicts with an access (A_2) by another processor (P_2) and (ii) A_2 is ordered after A_1. Since A_1 is a Reordered access, it does not get squashed by A_2. Figure 2 shows an example for TSO, where A_1 is *rd y* and A_2 is *wr y*.

This is a necessary but not sufficient condition for a cycle. A cycle (for two processors) additionally needs that P_2 issues a subsequent access (A_3) that conflicts with an earlier access from P_1 (A_4) and is ordered before A_4. This is shown in Figure 2, where A_3 is *rd x* and A_4 is *wr x*. We have an SCV.

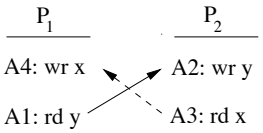

Figure 2: Example of a pattern that creates an SCV.

SCsafe may need to undo Reordered accesses to avoid SCVs. Hence, performing a Reordered store in RC should not update memory irreversibly. Instead, it can update a speculative cache or, as in the design we present later, use exclusive prefetch to obtain exclusive cache permissions — the access updates the cache later when it is not Reordered anymore.

4.2 Basic SCsafe Operation

From the previous discussion, we can deduce the low-cost approach that SCsafe uses to detect SCVs: SCsafe stalls accesses that conflict with a Reordered access in another processor. In most cases, the stall will naturally go away as accesses finish. However, if an SCV is about to occur, the participating processors will necessarily deadlock. At that point, SCsafe records the SCV, breaks the deadlock, recovers the SC state, and resumes execution transparently to the running program. We now consider each step, starting with the stall.

To stall accesses, SCsafe proceeds as follows:

• When an access (A) in a processor (P) becomes Reordered, SCsafe's hardware places the address accessed by A, A's program counter, and whether A is a load or a store in a structure in P's cache controller called the *Reordered Set* (RS).

• Coherence transactions received by P's cache are checked against its RS for an address match (at the cache line granularity for realistic hardware). Specifically, incoming reads are checked against the writes in the RS, while incoming writes are checked against both reads and writes in the RS. If there is a match, the transaction is refused (i.e., answered with a Nack), which will cause the requester to retry.

• When A ceases to be Reordered, SCsafe's hardware removes it from the RS; it cannot trigger SCVs anymore.

If a processor runs out of RS entries, it stalls. Also, note that, while an address is in a processor's RS, the local cache has to observe all the external coherence transactions directed to the corresponding line. Hence, we need to carefully handle cache evictions of lines with RS entries. If the line is clean, it can be evicted silently, since future coherence transactions will still be observed locally (in directory protocols, because the directory has not been notified; in snoopy protocols, because invalidations are broadcasted).

If, however, the evicted line is dirty *and* the machine uses directory-based coherence, special care is needed, since the visibility of future coherence transactions is in jeopardy. In this case, SCsafe rolls-back to the state before the instruction that created the RS entry. We will see how this is done. In practice, cache replacement algorithms that follow LRU-like policies rarely choose to evict a line with a recently-inserted RS entry.

Consider Figure 2 again. Assume load A_1 performs before store A_4 finishes, and address y is placed in P_1's RS. Later, store A_2 executes, initiating a coherence transaction that reaches P_1's cache and hits in the RS. The transaction is nacked, preventing P_2 from executing A_2. When store A_4 finishes, address y is removed from P_1's RS. A retry of store A_2 by P_2 now succeeds. However, if the timing is such that A_2 waits on A_1, and A_4 waits on A_3, an SCV has just been avoided, and the system deadlocks.

Section 5.1 describes the RS operation in detail.

Nacking requests simplifies SCV handling: the SCV has been avoided at the last minute, incorrect data has not been consumed, and execution can be easily rolled-back. Nacking [14] has been implemented in several multiprocessors, including DASH [17] and the Silicon Graphics Origin [16].

4.3 Types of Stalls

SCsafe's stalls can be classified based on whether or not they cause deadlocks (Table 1). In most cases, a stall is temporary. It goes away after an access completes and what used to be its Reordered accesses are removed from the RS. These stalls do not flag SCVs (Table 1).

	No Deadlock	Deadlock	
		True Dependences	False Sharing
SCV?	No	Yes	No

Table 1: Deadlocks versus SCVs in SCsafe.

When stalls cause a deadlock, we have N processors stalling one another in a cycle — each processor waiting on another processor's RS. If the cross-processor dependences are all true dependences, an SCV has been averted. Examples of two- and three-processor cycles of this type under RC are shown in Figures 3(a) and (b), respectively. In the figures, the addresses in the RS are shown in a box, and a nacked access is represented by an arrow that curves back to its source.

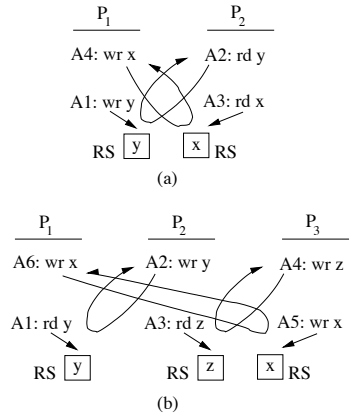

Figure 3: Examples of deadlocks caused by SCVs.

If at least one of the cross-processor dependences is due to false sharing, the deadlock is not due to an SVC (Table 1). Since processors initiate coherence transactions at cache-line granularity, requests are nacked even though the accesses are to different words of the same line. This is shown in Figure 4(a), where words a and b share a line. As we will see, SCsafe detects this case, breaks the deadlock, restores SC, and continues without recording any SCV.

When there is a deadlock (due to true dependences or false sharing), it is possible that a processor that does not participate in the cycle also gets embroiled in the deadlock. This occurs when the processor accesses an address that is already part of a cycle in other processors. An example is processor

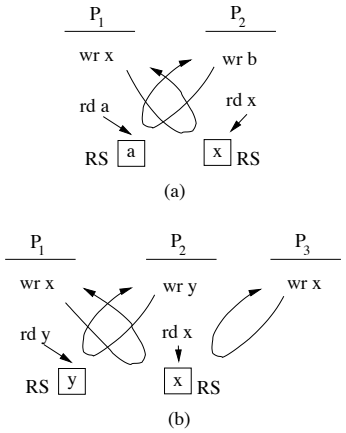

Figure 4: Other cases of deadlocks.

P_3 in Figure 4(b). In this case, when SCsafe breaks the deadlock (Section 4.6), the processor is released.

Overall, SCsafe is *precise* because of two reasons. First, SCsafe records all the SCVs that occur (for a given dynamic execution of the program). This is because all SCVs cause deadlocks. Second, SCsafe only records true SCVs. The reason, as we will see, is that SCsafe identifies the deadlocks caused by false sharing, and silently recovers from them.

4.4 Detecting a Deadlock

A sign that a processor P_i may be participating in a deadlock is that its RS bounces an external request, and one of its own requests is being bounced by another processor. However, SCsafe initiates the *Deadlock Detection and Analysis* (DDA) algorithm in P_i only if and when it is P_i's *oldest* unfinished access (A_{old_i}) the one bounced by another processor. In TSO, A_{old_i} is the write at the head of the write buffer; in RC, A_{old_i} is either such a write or, if the write buffer is empty, the read at the head of the ROB.

At a high level, when P_i bounces an external request and its A_{old_i} access is also being bounced, SCsafe embeds some information in P_i's future A_{old_i} retry messages. Such information will propagate to all of the processors involved in the potential cycle. If the information ever reaches back to P_i, it means that there is a cycle. Then, P_i records the local state of the cycle and adds further information to its retry messages. When this additional information reaches back to P_i again, it means that all of the processors in the cycle have recorded their state. Then, P_i initiates recovery.

The information included in a retry message is: (1) two bitmaps with as many bits as processors in the machine, which record the processors participating in the cycle (*Round0* and *Round1*); (2) the byte offset and datatype (such as word, halfword, etc) of the data accessed by A_{old_i} in the line *OffType*); and (3) a bit that records whether the cycle is due to false sharing (*FS*). All request messages contain these fields, which are 5 bytes long for the systems we evaluate, but they are ordinarily unused.

DDA starts when P_i bounces an external request and its A_{old_i} access is bounced. At this point, the local SCsafe hardware includes the following in P_i's future A_{old_i} retry messages: (1) *Round0* with bit *i* set, (2) an empty *Round1* bitmap, (3) *OffType* set to the byte offset and datatype of the data accessed by A_{old_i} in the line, and (4) *FS* set to zero.

The processor at the receiving end (P_j) simply ignores this information if its own A_{old_j} is not being bounced. However, if it is, the SCsafe hardware performs two actions. First, it checks if the *OffType* in the incoming message matches an address in P_j's RS exactly, or only because of false sharing. If the latter is true, P_j sets a local FS bit. Second, P_j includes in its own A_{old_j} retry message: (1) *Round0* coming from P_i augmented by also setting the *j* bit, (2) *Round1* coming from P_i; (3) *OffType* with the byte offset and datatype of the data that A_{old_j} accesses, and (4) the FS bit coming from P_i OR-ed with the locally-generated FS bit.

Successive processors in the cycle perform the same two actions. If there is a cycle, P_i eventually finds out that it is bouncing an incoming request with P_i's own ID bit already set in *Round0*. Hence, we have a deadlock. P_i also computes its local FS bit. If the incoming message's FS bit or the local FS bit is set, the cycle is due to false sharing. In this case P_i simply initiates recovery (Section 4.6).

Otherwise, the cycle is due to true dependences. Then, P_i records the local SCV state (Section 4.5). In addition, in future retries of A_{old_i}, in addition to including the usual information, P_i sets the *i* bit in *Round1*. The same operation is performed by all the other processors in the cycle. Therefore, a second wave of information traverses the cycle. Finally, when P_i finds that it is bouncing an incoming request with P_i's own ID bit already set in both *Round0* and *Round1*, it knows that all the processors in the cycle have recorded their local SCV state. Then, P_i initiates the recovery.

Note that it is possible that multiple processors in a given cycle start the DDA algorithm at the same time; the algorithm works equally well. In this case, multiple processors may initiate the recovery. However, each processor in the cycle logs the local SCV state *only once*. Section 5.2 describes the DDA algorithm in detail.

Figure 5 shows an example of DDA for four deadlocked processors due to true dependences. In the figure, only P_0 initiates DDA. For each message, the lighter bitmap is *Round0*, while the darker one is *Round1*. The numbers in parenthesis show the temporal sequence of events. While each processor continuously issues retry messages, each chart only shows one retry per processor. In Chart (a), as information is propagated from P_0 to back to P_0, each processor populates *Round0*. In Chart (b), each processor finds its bit set in *Round0*, logs the SCV state, and populates *Round1*. Finally, Chart (c) shows what happens after P_0 has received message (8), as will be explained in Section 4.6. In this case, P_0 initiates recovery (9) and, as a result, the next retry from P_3 succeeds (10).

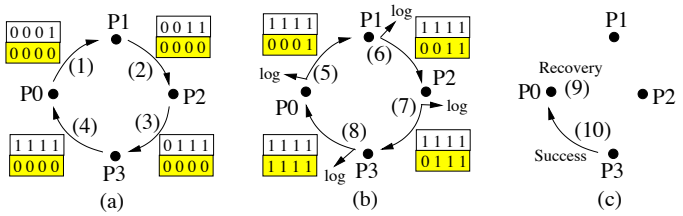

Figure 5: Example of deadlock detection and analysis.

4.5 Recording the SCV

As indicated above, when a processor P_i bounces for the

first time an incoming access that contains *Round0* with bit *i* set and *Round1* with bit *i* clear, and there is no false sharing, SCsafe records the local SCV state. Specifically, SCsafe dumps four pieces of information into a memory in the cache controller: the program counter (PC) and the address accessed by the two local instructions involved in the cycle. One of the instructions is A_{old_i} (the access being bounced, such as A_4 in Figure 3(a)). Its PC and address are readily available. The other instruction is the one that created the RS entry that bounces the incoming coherence request (such as A_1 in Figure 3(a)). We identify the RS entry as the one that exactly matches the address in the incoming bounced request. RS entries contain both the address and the PC.

A processor's recording operation is unlikely to take more than several tens of cycles and, therefore, has a negligible performance impact.

4.6 Recovery while Retaining SC

When a processor P_i bounces an incoming request where bit *i* is set in both *Round0* and *Round1*, P_i initiates recovery. The goal is to return the deadlocked processors to production execution transparently and right away. The recorded SCV information can be analyzed off-line later.

To understand the recovery, note that the state of the global memory system at this point is the one before A_{old_k} for all the *k* processors participating in the deadlock. This is because, as we explain later, Reordered stores in RC do not update memory, but obtain exclusive cache permissions. However, the pipeline state of each processor *k* is beyond this point. Hence, to recover from the deadlock while retaining SC, we need for at least one of the *k* processors (e.g., P_i) to roll back its pipeline state to when its bounced request (A_{old_i}) was at the head of its ROB, it had no Reordered accesses, and its RS was empty. An empty RS allows other deadlocked processors to make progress. Concurrently, P_i can re-execute A_{old_i} and subsequent instructions. This approach is attractive because it only requires logic that is local to the processor, and is compatible with common pipeline-recovery mechanisms.

Specifically, recovery in a processor involves rolling back all the instructions that have been retired from the ROB since the still-unfinished A_{old} access. These instructions can be of all types, and may include stores. To roll back, SCsafe uses a History Buffer (HB) circular queue [31], which has been used for recovery in previous proposals (e.g., [13, 25]) with different designs. SCsafe uses an HB design that can have multiple retired stores. Recall that the HB temporarily stores the processor state that each of the retired instructions overwrites. As an instruction retires from the ROB, if there are any preceding unfinished accesses (i.e., stores), SCsafe fills an entry at the tail of the HB. Each entry contains, for the register that the instruction overwrites, the old value and the old register mapping. It also includes the instruction's PC.

SCsafe does not store speculatively generated state in the caches. A Reordered store in RC keeps its state in the write buffer, and triggers an exclusive prefetch to the cache, to bring the corresponding line in Exclusive state into the cache. When the prefetch completes, the store is considered performed, and is entered in the RS. In this way, when stores are eventually merged in order with the memory system, they can do so very quickly, while their rollback before that point is simple. The actual recovery process under TSO and RC is described in Section 5.3.

4.6.1 Livelock Considerations

In the example of Figure 5, P_0 rolls back in Step (9). This operation clears its RS, which enables the access from P_3 to succeed (Step (10)). As P_3 makes progress, P_2 and P_1 will also make progress.

Depending on the timing, it is possible that multiple processors in the same deadlock cycle (or all of them) perform rollbacks concurrently. The algorithm works correctly. As processors re-execute their A_{old} accesses again, we prevent them from getting into the same deadlock again by disallowing any reordering during A_{old} execution. After A_{old} finishes, reordering is enabled again. In this way, processors make guaranteed forward progress.

4.7 Store Atomicity Considerations

In SCsafe, we use an RC implementation with atomic stores (also called multiple-copy atomic stores). This means that a store operation by a processor can be observed by another processor only if it has been made visible to all other processors, and that stores to the same location are serialized.

Store atomicity is required by well-known RC instantiations, such as those of Tilera [32], SPARC RMO (and PSO) [1], and Alpha [30]. Hence, the SCsafe design presented here is directly applicable to all of them. Store atomicity is not required by the IBM Power [23] or ARM [2] models — although it is likely that, in practice, many (or most) ARM implementations enforce it, due to their simple memory hierarchies.

Relaxing store atomicity would complicate SCsafe nontrivially. It would allow a processor P_1 with a pending store *S* to provide *S*'s value to another processor P_2 before P_1 receives the response to *S*'s transaction. If such a response was a nack, it would be necessary to recall the value from P_2. We leave an extension for non-atomic stores to future work.

5. SCsafe IMPLEMENTATION

We now detail the operation of three components of SCsafe: Reordered Set (RS), DDA, and History Buffer (HB). Then, we examine SCsafe's hardware complexity.

5.1 RS Implementation and Operation

The RS is a hardware structure in the cache controller that stores the addresses accessed by the processor's current Reordered references. Each entry contains the address, the PC of its instruction, and some additional state. New entries are dynamically added and removed. The RS is organized as a circular FIFO queue, ordered in program order of the Reordered accesses. In this section, we describe its operation under TSO and under RC.

5.1.1 Operation under TSO Hardware

Given an unfinished store, its Reordered accesses are all the loads that follow it in program order, up to (but not including) the earliest not-yet-performed load. An unfinished load cannot have any Reordered accesses.

From this discussion, the RS can only contain loaded addresses. Moreover, when a store finishes, we need to remove from the RS the addresses of all the loads that follow the store and that precede the next unfinished store. Hence, to speed-up RS operation, we design SCsafe as follows. Each instruction in the ROB has a Write Tag (WT). When a store is inserted in the ROB, its WT is set to the value of the previous instruction's WT plus one. For non-store instructions, the WT is that of the previous instruction. Hence, a store plus all the instructions following it until (but not including) the next store have the same WT. The WT is also stored in each RS entry.

The algorithm to insert entries in the RS tail and remove them from the RS head is as follows. When a load (l_1) performs and (i) there is at least one preceding store that is unfinished (it can still be in the ROB or already retired in the write buffer) and (ii) all preceding loads in the ROB are performed, then:

- The address loaded by l_1 is inserted in the RS.
- The addresses loaded by all the loads l_n that follow l_1 in program order up to (but not including) the earliest not-yet-performed load are also inserted in the RS in program order.

Entries are removed when a store finishes. In this case, starting at the RS head and moving toward the tail, SCsafe removes all the loads that have the same WT as the store. The process stops when the RS is empty or when we find the first load with a higher WT (which follows a subsequent unfinished store).

5.1.2 Operation under RC Hardware

Given an unfinished store, its Reordered accesses are all the performed loads and all the performed stores (i.e., those that have completed the exclusive prefetch) that follow it in program order. Given an unfinished load, its Reordered accesses are all the performed loads that follow it in program order.

Hence, the RS can have both loads and stores. Also, any performed access that is preceded by at least one unfinished access needs an RS entry. These facts make the hardware costlier, but the insertion and removal algorithms simpler. Indeed, as a load or a store access *A* enters the ROB, if there is at least one unfinished earlier access, SCsafe reserves an empty entry for *A* at the tail of the RS. Later, this entry may be filled when *A* performs, and the entry may be removed when an earlier access finishes.

Specifically, when an access *A* performs, if there is at least one unfinished earlier access, then SCsafe stores *A*'s information in its reserved RS entry. Otherwise, *A* is not a Reordered access and, hence, has no RS entry.

When an access *A* finishes, if there is at least one unfinished earlier access, the RS entry for *A* is left unchanged. Otherwise, *A* was the earliest unfinished access and, hence, had no RS entry. In this case, SCsafe may need to remove RS entries. Specifically, starting at the RS head and moving toward the tail, SCsafe removes the RS entries of all the finished accesses until it reaches the entry for the first (i.e., earliest) unfinished access. This entry is also removed, since its access is not Reordered anymore. Note that this entry will still be empty if the corresponding access has not yet performed.

5.2 The DDA Algorithm

When the A_{old_i} access of a processor P_i is being bounced, and P_i's RS bounces an incoming request, the local SCsafe hardware runs the DDA algorithm of Figure 6. If the incoming request does not any contain deadlock information (Line 1), then P_i starts deadlock detection by including, in its A_{old_i} retries: *Round0* with the single bit *i* set, a null *Round1*, the byte offset and datatype of A_{old_i} ($OffType_i$), and a clear FS bit (Line 2).

```
1  if (incoming request has no info)   /* this proc starts DDA */
2    Include in A_old_i msg (i, null, OffType_i, 0)
3  else {      /* this proc does not start DDA */
4    if (hit due to false sharing)
5      local_FS =1
6    if (i bit is not set in incoming Round0){
7      /* this proc hasn't informed all the other procs in the cycle about its participation */
8      Include in A_old_i msg (Round0 |= i, null, OffType_i, FS |= local_FS)
9    }
10   else {  /* information has propagated around the cycle */
11     if (FS | local_FS == 0) {  /* cycle with only true dependences */
12       if (i bit is not set in incoming Round1) {  /* only 1st round completed */
13         Record SCV              /* record the local SCV state */
14         Include in A_old_i msg (Round0, Round1 |= i, OffType_i, 0) /*start 2nd round*/
15       }
16       else {    /* 2nd round completed now */
17         Recover (P_i)    /* start recovery for proc P_i, which breaks the cycle */
18       }
19     }
20     else {  /* false sharing cycle; first detection */
21       Recover (P_i)    /* breaks the cycle; no recording of SCV */
22     }
23   }
24 }
```

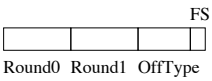

Figure 6: The DDA algorithm, as executed by P_i.

Otherwise, deadlock detection is already in progress (Line 3). In this case, P_i first checks if it is bouncing an access due to false sharing (Line 4) and, if so, sets the *local_FS* bit (Line 5). Moreover, if bit *i* is not yet set in the incoming *Round0* (Line 6), it means that P_i has not yet informed all the other processors in the potential cycle about P_i's participation in the cycle. Hence, SCsafe takes the deadlock information in the incoming bouncing message, augments it, and includes it in future A_{old_i} retries. This augmentation involves setting bit *i* in *Round0*, keeping *Round1* null, enclosing the byte offset and datatype of A_{old_i}, and OR-ing the local_FS bit to FS (Line 8).

If bit *i* is set in the incoming *Round0* (Line 10), we have a cycle and the information has propagated around the cycle. SCsafe first checks if any processor (including P_i) detected false sharing (Line 11). If so (Line 21), P_i recovers. Otherwise, SCsafe checks if the information has gone around the cycle once or twice. If the former, SCsafe records the local SCV state (Line 13) and augments the retry messages by setting bit *i* in *Round1* (Line 14). If the latter, since all processors have recorded the SCV, SCsafe initiates the recovery (Line 17).

5.3 HB Operation and Recovery

In our conservative design, as an instruction retires from the ROB, if there are any preceding unfinished accesses (which are necessarily stores in our model), SCsafe fills an entry at the tail of the HB. Therefore, the retirement of a store forces subsequent instructions to fill HB entries. In addition, when a store finishes, if there is no earlier unfinished store, the

hardware removes HB entries. Specifically, starting at the HB head, it walks toward the tail, freeing all the entries until (and including) the entry for the next unfinished store — or until the HB is empty.

In RC, some of the freed entries may correspond to finished stores. Those can now proceed to update the cache; hopefully, the exclusive prefetches have already brought the lines to the cache, and the stores can drain immediately. Both in TSO and RC, the next unfinished store can now also proceed to update the cache — some of its latency may be hidden by an exclusive prefetch issued earlier that has not yet completed. Recall that, under TSO, the stores have to be merged in program order.

The HB interacts with deadlock recovery as follows. Sometimes, the oldest unfinished access (A_{old}) ends up stalled in a deadlock, while being followed by Reordered accesses. To recover, SCsafe needs to undo all the Reordered accesses.

This A_{old} access with Reordered accesses is handled differently in TSO and in RC. In TSO, the A_{old} is a retired store at the head of the write buffer, and its Reordered accesses are loads; the HB will have entries for all these Reordered loads. In RC, the A_{old} can be a store or a load. If it is store, it is a retired store at the head of the write buffer, and its Reordered accesses can be loads and stores; the HB will have entries for all of them. However, if A_{old} is a load, it is the earliest unretired load and its Reordered accesses are only loads; there are no entries in the HB. In this case, there is no interaction with the HB. In the following, we explain how we recover in TSO and in RC.

5.3.1 Recovery in TSO

Recovery for a processor starts by first clearing the RS, the ROB, and the whole write buffer except for its earliest entry. Then, starting from the HB tail and walking toward the head, each HB entry is used to undo the state changes performed by one instruction. After the whole HB is applied, all the Reordered accesses have been undone, and the processor has the state at the point of performing A_{old}. The hardware simply attempts to perform the A_{old} access again. After it succeeds, it starts fetching again.

5.3.2 Recovery in RC

If A_{old} is a store, the Reordered accesses can include other stores. Recall that these stores had been left in the write buffer without being merged with memory, while an exclusive prefetch was sent to the cache. Consequently, if A_{old} is a store, recovery proceeds as in TSO.

If A_{old} is a load, A_{old} is still in the ROB and the write buffer is empty. The HB is empty and there is no HB to apply. Recovery involves clearing the RS and flushing the instructions in the ROB that follow A_{old}. The hardware simply attempts to perform the A_{old} access again.

5.4 Hardware Complexity

The hardware required by SCsafe is of modest complexity, especially when compared to other SCV detection schemes such as Vulcan [21] and Volition [24]. It has three components: the RS, the DDA mechanism, and the HB (Figure 7).

The RS is a circular FIFO queue in the L1 cache controller. Each entry has an address, a R/W bit, a PC and, under TSO, a WT. Entries are allocated when accesses become Reordered,

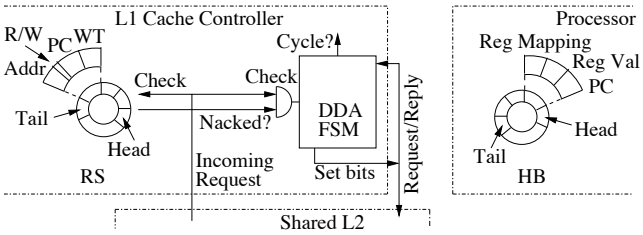

Figure 7: SCsafe hardware.

and deallocated when they cease to be Reordered. Incoming requests are compared to the RS addresses. For efficiency, the RS is not implemented as a CAM. Instead, we perform sequential comparisons, 4 entries at a time. This is reasonable because this operation is not time-critical, and because, often, few RS entries are full — e.g., the RS evaluated in Section 7 has 32 entries, but it fills on average only 6.3. A further optimization involves using a Bloom filter.

The DDA mechanism consists of an FSM in the L1 cache controller that examines some incoming messages and updates some outgoing ones. Specifically, it reads information from incoming messages that bounce off the RS, and sets some bits in outgoing retry messages. Such bits are two processor bitmaps, information to identify which byte offset of the line was accessed and its datatype, and an FS bit. For a 16-processor machine with 32-byte cache lines, this amounts to 5 bytes. All request messages now include these bits.

Note that the DDA does not add any new messages. It simply tags existing messages. Hence, it does not need any additional virtual channels. Moreover, each DDA FSM operates independently and can declare a cycle locally.

Nacking (or bouncing) a request simply means that a request failed, and the FSM at the sender is informed that it needs to retry. It is a null transaction that had no side-effects. Hence, it does not impose any restriction on the coherence protocol. Beyond request nacking and the extra bits per request, there is no other change to the coherence protocol: no new messages, new states, or new transactions. There are no changes to the directory module.

Finally, each processor has a circular HB to recover from Reordered accesses in a deadlock. Each HB entry has a register value, a register mapping, and a PC. An HB entry is filled quickly with minimal computation, although it requires a register read. No speculative updates go to L1 caches. Rollback involves undoing one instruction at a time, but it happens rarely.

Overall, the SCsafe hardware is mostly local to each processor node and, in part, uses known recovery techniques.

6. EVALUATION SETUP

In our evaluation, we use detailed cycle-level execution-driven simulations using the SESC simulator [26]. We evaluate SCsafe's ability to detect SCVs in parallel programs running under RC or TSO. We also evaluate SCsafe's performance overhead. We model a multicore with 16 cores connected in a mesh network with a directory-based MESI coherence protocol. Each core has a private L1 cache and a bank of a shared L2 cache. The RS stores word addresses. Table 3 shows the architecture. From the table, we see that the storage needed by the SCsafe hardware is modest.

To evaluate SCsafe's ability to detect SCVs, we use a set

Code	RC					TSO				
	SCsafe		IF	IF-CoV		SCsafe		IF	IF-CoV	
	#SCVs	#Stalls	#Squashes	#Timeouts	#Stalls	#SCVs	#Stalls	#Squashes	#Timeouts	#Stalls
bakery	3	4494	5630	254	6647	3	4362	4583	6	6980
dekker	14	91412	76471	29	60961	17	83093	85603	21	58183
harris	302	23256	25885	2012	32792	191	24010	21723	1679	33210
lazylist	162	8845	8840	1039	9105	75	7946	8166	798	9466
takequeue	165	6980	6856	993	6731	98	6905	6816	788	6319
aharr	100	11525	11593	859	11494	74	10546	11504	803	11602
moirbt	218	9373	10381	1293	9459	143	8775	10893	1015	8522
moircas	149	5648	7964	843	9667	35	5225	6189	616	10833
ms2	193	19039	21907	1509	23244	145	17676	20831	1102	22837
snark	10	9431	14636	143	15855	13	9786	13262	180	17495
msn	2	8322	7302	35	6715	0	7676	7829	26	6222
mst	3	7927	9527	28	10663	0	7765	8230	21	12102
Average	110	17188	17249	753	16944	66	16147	17136	588	16981

Table 2: SCV detection for the kernel programs.

Architecture	16-core chip multiprocessor (CMP)
Core	Out of order, 3-issue wide, 2.0 GHz
ROB; wr. buffer	96-entry; 32-entry
L1 cache	Private 32KB WB, 4-way, 2-cycle RT
L2 cache	Shared 2MB WB, with 16 128KB banks Bank: 8-way, 11-cycle RT (local)
Line size	32 bytes
Cache coherence	MESI, full-mapped directory
On-chip network	4x4 2D-mesh, 5 cyc/hop, 256bit links
Off-chip memory	200-cycle RT
Reordered Set	32 entries/proc: 8B addr+1b R/W+8B PC+1B WT
History Buffer	64 entries/proc: 8B reg+2B map+8B PC
Retry delay	20 cycles before issuing a retry message
Record an SCV	5 cycles of visible overhead

Table 3: Architecture modeled. RT means round trip.

of 12 programs [5, 6, 10] that implement concurrency algorithms, such as a lock-free queue and a work-stealing queue. We remove the fences in these codes and, therefore, their execution may violate SC on plain RC or TSO hardware. We call them *kernels* (Table 4). They come with their inputs. Each thread of each kernel executes a loop with 200 iterations that access shared data structures.

bakery	Mutual excl. algorithm for any # of threads
dekker	Mutual excl. algorithm for two threads
harris	Non-blocking set
lazylist	Concurrency list algorithm
takequeue	Cilk THE work stealing algorithm
aharr	Variant of harris
moirbt	Non-blocking sync. primitives
moircas	Non-blocking sync. primitives
ms2	Two-lock queue
snark	Non-blocking double-ended queue
msn	Non-blocking queue
mst	Non-blocking queue

Table 4: Kernels of concurrency algorithms.

SCsafe detects and records SCVs precisely during the execution, and recovers from an SCV while retaining SC. We compare SCsafe to an SCV-detection scheme that, when an SCV is found, terminates execution because SC cannot be maintained. Examples of such a scheme are Vulcan [21] and Volition [24]. We also compare SCsafe to two SC-enforcing-only schemes: InvisiFence [4] with and without Commit on Violation (we call them IF and IF-CoV). Such conservative schemes squash execution as soon as a certain necessary condition for an SCV occurs. They are not usable to report SCVs because they would report many false positives (Section 2.2). IF-CoV uses a 4,000-cycle timeout threshold.

To evaluate the performance overhead of SCsafe over plain RC or TSO hardware, we use 16 SPLASH-2 [34] and PARSEC [3] programs. We call them *apps*. SPLASH-2 apps use the default inputs; PARSEC use simmedium. Apps run correctly on RC or TSO hardware, but SCsafe can induce performance overhead as it tries to conservatively enforce SC.

7. EVALUATION

7.1 SCV Detection

7.1.1 Number of SCVs Detected

To assess SCsafe's ability to detect and record SCVs, we run the fenceless kernels under RC and TSO. We report the number of SCVs and the number of accesses stalled. The *apps* are found to have practically no SCV, and so they are not shown. For comparison, we also run the fenceless kernels with IF and IF-CoV. Since these schemes cannot observe SCVs, we report the number of squashes (in IF), and stalls and timeouts (in IF-CoV). The data is shown in Table 2, where RC data is on the left and TSO data on the right.

Consider the RC environment. Column 2 shows the number of dynamic SCVs detected by SCsafe. We see that SCsafe detects SCVs in all the kernels. On average, it detects 110 SCVs. Column 3 shows the number of dynamic accesses stalled by SCsafe. Such number is more than 100 times higher than the number of SCVs. Most of these stalls are very short and unrelated to an SCV. This shows that seeing a single access reorder from another processor (e.g., a data race) is not a good SCV indicator; one needs to see a dependence cycle.

Column 4 shows the number of squashes in IF. This number is similar to the stalls in SCsafe — but not exactly the same because memory accesses interleave differently. It is, however, much higher than the number of SCVs. Finally, Columns 5 and 6 show the number of timeouts and stalls in IF-CoV. The number of stalls is also similar to SCsafe. The number of timeouts is closer to the number of SCVs, but it is still much higher for two reasons. First, as we will see, most timeouts are caused by false sharing. Second, when a group of processors times out, this counter increases by the number of timed-out processors. In any case, IF-CoV's timeouts are unable to record information useful to debug SCVs.

The data for the stricter TSO is similar, with fewer SCVs. Overall, we conclude that SCsafe successfully finds SCVs, and that conventional SC-enforcement approaches cannot be used for SCV detection and debugging.

7.1.2 Stopping versus Continuing

Unlike SCsafe, other precise SCV detection schemes such

as Vulcan [21] and Volition [24] terminate execution once they find an SCV. They are unable to retain SC execution and, therefore, they could find additional artificial SCVs caused by the non-SC execution. We call them *Stop* approaches. Debugging with them involves multiple iterations of: SCV detection, termination, fixing the SCV by inserting fences, and then re-execution from the beginning of the application. It usually takes several runs to detect the SCV bugs that SCsafe detects in a single run. Also, these schemes are incompatible with production runs.

We compare SCsafe to the operation of SCsafe with the Stop approach. In this case, each re-execution finds one SCV, which is fixed with fences. Table 5 compares the number of runs to detect all the SCVs in the kernels for the two approaches, using RC. This table differs from Table 2 in that we perform as many runs as needed to find all SCVs (Table 2 corresponds to only one run). We see that Stop typically requires several runs to find all the SCVs. SCsafe only needs one run or, in three kernels, two.

Code	SCsafe	Stop	Code	SCsafe	Stop
bakery	1	1	dekker	1	1
harris	1	6	lazylist	1	4
takequeue	1	6	aharr	1	7
moirbt	1	3	moircas	1	4
ms2	1	2	snark	2	14
msn	2	9	mst	2	8

Table 5: Number of runs to find all SCVs in RC.

7.1.3 Sensitivity Study

We examine the sensitivity of SCV detection to the size of the RS. As the size of the RS increases, the degree of reordering of memory operations increases, which leads to more SCVs. The results are shown in Figure 8 for both RC and TSO hardware. The figure shows the average number of SCVs observed per kernel for 16-processor runs.

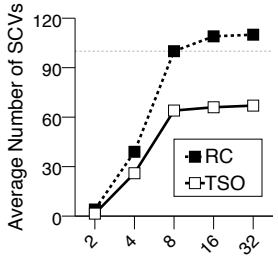

Figure 8: Impact of RS size on the number of SCVs.

In the figure, we change the RS size from 2 to our default of 32. We can see that, as the hardware becomes more aggressive, SCsafe detects more SCVs. Also, RC systems always detect more SCVs than TSO ones. Overall, we choose our default size based on when the curves saturate.

7.2 SCsafe Execution Time Overhead

Compared to conventional hardware, SCsafe incurs two types of execution overhead. The first is access stall overhead. It is mainly caused by accesses that hit in the RS of other processors and have to retry. It can also be caused by the HB or RS being full. The second overhead is recovery from deadlock. This operation requires restoring the architectural state by flushing the pipeline and traversing the HB.

Figure 9 shows the execution time of SCsafe for *apps* normalized to the execution time on plain RC hardware. The bars are labeled *S*. We also show bars for the IF-CoV scheme for SC enforcement. The bars are labeled *I*. The bars are broken down into categories. SCsafe has *Recovery* (overhead of accesses that deadlock, including their stall, recovery, and re-execution), *Stall* (overhead of stalls that do not deadlock), and *Useful* (rest of the time). IF-CoV has *Timeout* (overhead of accesses that timeout, including their stall, squash, and re-execution), *Stall* (overhead of stalls that do not timeout), and *Useful*. IF-CoV uses flash clear of dirty lines in a squash.

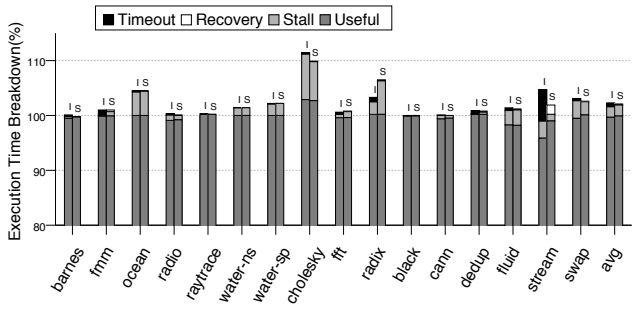

Figure 9: Execution time of *apps* with IF-CoV (I) and SCsafe (S) on RC. The bars are normalized to plain hardware.

The figure shows that, on average, SCsafe induces an overhead of ≈2% over RC. The overhead of IF-CoV is similar. We see that most of the SCsafe overhead is due to stall cycles. The stall is small because the latency of nacked accesses is partially hidden by the execution of other instructions. A few codes have a larger stall time. Typically, these are codes with fine-grain sharing, where the stalls are due to false sharing. In Radix, SCsafe sometimes stalls because the HB is full. The figure also shows that recovery and timeout cycles are only significant in one application. This is because there are very few dependence cycles in these codes. Overall, SCsafe induces a tiny overhead, which is an acceptable cost to ensure SC. A similar result can be shown for TSO.

In prior work, the SC++ scheme [13] used HBs to enforce SC. They found that they needed 512-entry HBs to hide reordered accesses, while we use 64-entry HBs and only observe modest HB-full stall in a couple of applications. The reason is that they model a distributed shared-memory architecture, with an order-of-magnitude higher cache to cache transfer latency than in our CMP.

We now consider the kernels. Since we removed the fences from these codes, they may run incorrectly on plain RC or TSO hardware. Hence, we only compare the execution time of SCsafe to IF-CoV. Figure 10 shows the execution time of the kernels for IF-CoV (labeled *I*) and SCsafe (labeled *S*) on RC. The bars are normalized to IF-CoV and broken down as above.

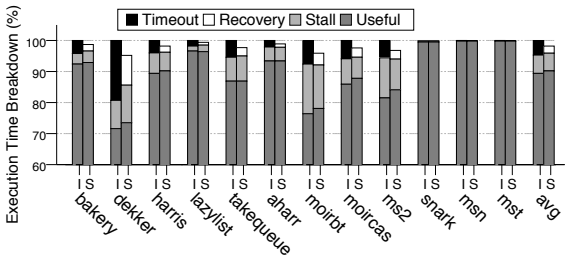

Figure 10: Execution time of kernels with IF-CoV (I) and SCsafe (S) on RC. The bars are normalized to IF-CoV.

With plain RC or TSO hardware, access reordering by the hardware would cause SCVs. With SCsafe, it causes stalls and recoveries. The figure shows that, on average, the stall cycles in SCsafe are about 6%. Recovery time is also visible. With IF-CoV, we see stalls and timeouts. On average, SCsafe has approximately the same execution time as IF-CoV. A similar result can be shown for TSO. Overall, therefore, the key capability of SCsafe, namely continuous and precise detection and recording of SCVs while enforcing SC does not come at the expense of any slowdown relative to an SC-enforcing-only scheme such as IF-CoV.

7.3 SCsafe Characterization

Table 6 characterizes SCsafe for all the programs on RC. We do not show a characterization on TSO due to lack of space. Columns 2-3 show the average and maximum number of entries used in the RS during execution. On average, the RS size is only around 6 entries for both kernels and *apps*. It can be shown that the corresponding number for TSO is ≈3. Columns 4-5 consider the reads and writes that are bounced due to a hit in an RS. The columns show, in order, the number of such accesses per 10K accesses, and the average number of cycles between the first bounce at an RS entry and the deallocation of that RS entry. As we can see, for the large majority of codes, the rate of bounced accesses is very low. In addition, the duration of the stall in an RS entry is only a few tens of cycles. The rate of bounced accesses does not correlate perfectly with the SCsafe stalls in Figures 9 and 10; other factors like the access rate or clustering have an effect as well.

Code	Reordered Set (RS) Size		Bounced Reads & Writes		Recovery Reads & Writes	
	Avg	Max	#/10K	Cyc.	#/10K	FS(%)
bakery	3.0	32	489.8	45.0	32.5	87.3
dekker	11.0	32	270.3	42.2	11.1	92.2
harris	6.7	19	30.3	69.7	2.0	79.5
lazylist	2.7	16	33.8	20.0	2.0	81.4
tkqueue	5.0	32	156.0	33.1	5.4	68.0
aharr	2.0	15	30.8	35.6	0.4	77.4
moirbt	8.5	32	138.2	54.7	6.6	91.2
moircas	7.2	32	124.8	34.5	3.8	93.9
ms2	3.2	27	297.9	34.4	16.2	73.8
snark	5.5	12	0.3	26.6	0.0	89.4
msn	12.0	30	7.2	39.2	0.0	94.5
mst	8.2	19	13.2	17.2	0.0	69.6
Avg.	6.2	24.8	132.7	37.6	6.6	83.2
barnes	10.7	32	2.3	40.7	0.0	100.0
fmm	6.2	32	2.2	71.1	0.1	96.9
ocean	7.0	32	1.0	62.6	0.0	100.0
radio	6.5	32	0.2	81.3	0.0	100.0
raytrace	10.5	28	2.9	25.8	0.0	100.0
water-ns	5.2	30	0.0	87.2	3.1	100.0
water-sp	10.5	32	0.2	76.5	0.0	100.0
cholesky	5.5	24	35.1	65.8	0.0	100.0
fft	10.5	32	3.0	55.2	0.0	100.0
radix	2.2	28	7.8	43.2	0.1	100.0
black	2.0	6	0.0	26.6	0.0	100.0
cann	6.5	13	0.5	39.1	0.0	100.0
dedup	0.7	24	105.8	82.0	0.7	100.0
fluid	6.2	30	0.3	25.8	0.0	100.0
stream	11.5	32	150.8	33.0	5.8	100.0
swap	0.5	32	0.3	42.6	0.0	100.0
Avg.	6.3	27.4	19.5	53.6	0.6	99.8

Table 6: Characterization of SCsafe on RC.

Columns 6-7 consider the reads and writes that are involved in a cycle and trigger a recovery. The columns show the number of such accesses per 10K accesses, and the percentage of such cycles caused by false sharing. We can see that recoveries are much rarer than bouncing events: on average, 20x rarer in kernels and 32x in *apps*. In addition, most of the dependence cycles in the kernels (83% on average) and practically all of those in the *apps* are due to false sharing. Hence, supporting a precise scheme like SCsafe is crucial. Finally, it can be shown that the traffic increase due to SCsafe is negligible.

7.4 SCsafe Scalability Analysis

Figure 11 shows SCsafe's execution time overhead as we change the processor count for the *apps*. Due to lack of space, the plot only shows a sample of the *apps*; however, the average corresponds to all of the apps. For each *app*, we show the overhead of SCsafe over plain RC hardware for 8, 16 and 32 processors, and over plain TSO hardware for 8, 16 and 32 processors.

The figure shows that, with increased numbers of processors, the average overhead of SCsafe does not change much, and stays around 2% for both RC and TSO. This shows that SCsafe is scalable. The actual changes in the bars with the number of processors are often very small. Moreover, they are also affected by the scalability of the baseline plain RC and TSO execution with the processor count.

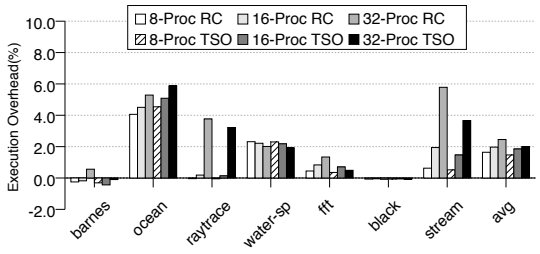

Figure 11: Scalability of SCsafe.

8. RELATED WORK

Section 2.2 already compared SCsafe to its most related schemes, namely Vulcan [21] and Volition [24]. Another related work is that of Chen et al. [8], who attempt to find hardware design bugs by checking for consistency-violating dependence cycles. In their work, a processor tags its cache lines with access IDs, which it then uses to record cross-processor dependences. It regularly sends this dependence information to a Centralized Graph Checker that checks for cycles. The hardware is centralized and likely intrusive.

There are schemes for SC-enforcement only, such as Load Speculation [11], Speculative Retirement [25], SC++ [13], BulkSC [7], ASO [33], InvisiFence (IF) [4], and Conflict Ordering (CO) [18]. These schemes are not usable for our purpose, namely to detect and record SCVs. Their goal is to steer execution away from any potential SCV when a (conservative) necessary condition for SCV is detected.

The condition that most of these schemes detect is the presence of an access that is observed while it is M-speculative relative to SC. The access would not be squashed because it is not M-speculative relative to the relaxed-consistency model supported by the machine. The access can be a load or a store. For example, in Figure 1(b), the access is store A_2.

When the condition is detected, the schemes squash at least the access that is M-speculative relative to SC. Most

schemes extend the speculation beyond the ROB, using history buffers, speculative caches, and checkpoints ([4, 7, 13, 25, 33]). The scheme IF with CoV waits for a time-out period before squashing.

CO [18] is different in that the condition that it looks for is two or more concurrent data races. This is a stronger condition than before, but one that can still cause false positives. To make a decision, a processor must first fetch from a global structure that has a list of pending writes. If the condition is true, CO squashes local accesses.

End-to-end SC [29] is a technique to enforce SC in a different way. It allows reordering for accesses to private locations, and enforces program order for memory accesses to shared locations. SCsafe is more aggressive in that it allows reordering of shared data accesses.

9. CONCLUSION

While there are prior proposals for SCV detection, they are limited in that they terminate program execution after detecting the first SCV because the program is now non-SC. Therefore, they cannot be used in production runs, wherein some SCVs may occur. In addition, they rely on complicated hardware.

To address this challenge, this paper presented *SCsafe*, the first architecture for relaxed consistency machines that detects and logs SCVs in a continuous manner, while retaining SC. In SCsafe, the processor hardware temporarily stalls incoming requests that conflict with some types of reordered accesses. A true SCV is detected when processors wait on each other in a cycle. In this case, SCsafe quickly detects the SCV, records it, recovers the processors' state, and resumes execution while retaining SC. As a result, it can be used in production runs. In addition, SCsafe is precise in that it identifies only true SCVs — cycles due to false sharing are discarded. Also, its hardware is simpler because it is mostly local to each processor, and uses known recovery techniques.

We evaluated SCsafe using simulations of 16-core multicores. In codes with SCVs, SCsafe detected and logged SCV bugs while enforcing SC during the complete execution. In codes with few SCVs, it added negligible slowdown. Finally, SCsafe was scalable with the number of processors.

10. REFERENCES

[1] "The SPARC Architecture Manual, V. 8," *SPARC International, Inc.*, 1992.

[2] ARM, *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R Edition Issue C*, July 2012.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, October 2008.

[4] C. Blundell, M. M. K. Martin, and T. F. Wenisch, "InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors," in *ISCA*, June 2009.

[5] S. Burckhardt, R. Alur, and M. M. K. Martin, "CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models," in *PLDI*, June 2007.

[6] J. Burnim, K. Sen, and C. Stergiou, "Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models," in *TACAS*, July 2011.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, June 2007.

[8] K. Chen, S. Malik, and P. Patra, "Runtime Validation of Memory Ordering Using Constraint Graph Checking," in *HPCA*, Feb. 2008.

[9] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective Data-race Detection for the Kernel," in *OSDI*, February 2010.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *PLDI*, June 1998.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *ICPP*, August 1991.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors," in *ISCA*, June 1990.

[13] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?" in *ISCA*, June 1999.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2011.

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

[16] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *ISCA*, June 1997.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," in *ISCA*, June 1990.

[18] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient Sequential Consistency via Conflict Ordering," in *ASPLOS*, March 2012.

[19] 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.

[20] 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.

[21] A. Muzahid, S. Qi, and J. Torrellas, "Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically," in *MICRO*, December 2012.

[22] 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.

[23] Power.org, *Power ISA^{TM} Version 2.06 Revision B*, July 2010.

[24] X. Qian, B. Sahelices, J. Torrellas, and D. Qian, "Volition: Scalable and Precise Sequential Consistency Violation Detection," in *ASPLOS*, March 2013.

[25] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models," in *SPAA*, June 1997.

[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] J. Sevcik, "Safe Optimisations for Shared-memory Concurrent Programs," in *PLDI*, June 2011.

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

[29] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi, "End-to-End Sequential Consistency," in *ISCA*, June 2012.

[30] R. L. Sites, "Alpha Architecture Reference Manual," *Digital Press*, 1992.

[31] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," in *ISCA*, June 1985.

[32] Tilera, *Tile Processor User Architecture Manual Rel. 2.4*, Nov. 2011.

[33] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *ISCA*, June 2007.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, June 1995.