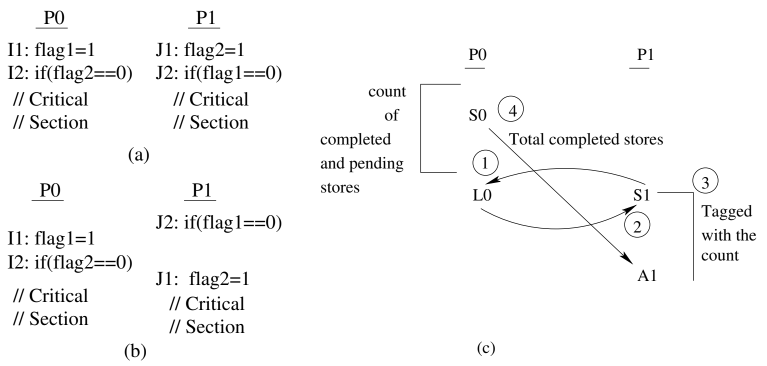
Hardware-Based Sequential Consistency Violation Detection Made Simpler

Mohammad Majharul Islam, Riad Akram, and Abdullah Muzahid
University of Texas at San Antonio
{*mohammadmajharul.islam, riad.akram, abdullah.muzahid*}*@utsa.com*

Abstract. Sequential Consistency (SC) is the most intuitive memory model for parallel programs. However, modern architectures aggressively reorder and overlap memory accesses, causing SC violations (SCVs). An SCV is practically always a bug. This paper proposes Dissector, a hardware software combined approach to detect SCVs in a conventional TSO machine. Dissector hardware works by piggybacking information about pending stores with cache coherence messages. Later, it detects if any of those pending stores can cause an SCV cycle. Dissector keeps hardware modifications minimal and simpler by sacrificing some degree of detection accuracy. Dissector recovers the loss in detection accuracy by using a postprocessing software which filters out false positives and extracts detail debugging information. Dissector hardware is lightweight, keeps the cache coherence protocol clean, does not generate any extra messages, and is unaffected by branch mispredictions. Moreover, due to the postprocessing phase, Dissector does not suffer from false positives. This paper presents a detailed design and implementation of Dissector in a conventional TSO machine. Our experiments with different concurrent algorithms, bug kernels, Splash2 and Parsec applications show that Dissector has a better SCV detection ability than a state-of-the-art hardware based approach with much less hardware. Dissector hardware induces a negligible execution overhead of 0.02%. Moreover, with more processors, the overhead remains virtually the same.

1 Introduction

Among various memory models, Sequential Consistency (SC) [15] is the most intuitive one. It guarantees a total global order among the memory operations where each thread maintains its program order. However, most commercial architectures sacrifice SC to improve performance. For example, x86 implements a memory model similar to TSO [30] which allows a later load operation to bypass an earlier store operation from the same processor. The overlapping and reordering of memory accesses can lead non SC behavior of a program, referred to as an *SC Violation* (SCV). Consider Dekker's algorithm in Fig. 1(a). Processor P0 first writes *flag1* (I1) and then reads *flag2* (I2) but P1 first writes *flag2* (J1) and then reads *flag1* (J2). Both flags are initially 0. In SC, either I2 or J2 will be the last one to complete. Therefore, either P0 finds *flag2* to be 1 or P1 finds *flag1* to be 1. It is even possible to have both flags to be 1 (e.g., if the completion order is I1, J1, I2, and J2). In any case, we can *never* have both flags to be 0. However, if the underlying memory model is TSO, it is possible for the load in J2 to bypass the store in J1 (Fig. 1(b)). As a result, the completion order becomes J2, I1, I2, J1 and both processors find the flags to be 0. The same problem can occur if I1 and I2 get reordered.



```
P0                    P1
I1: flag1=1           J1: flag2=1
I2: if(flag2==0)      J2: if(flag1==0)
 // Critical            // Critical
 // Section             // Section
                (a)
```

```
P0                    P1
                      J2: if(flag1==0)
I1: flag1=1
I2: if(flag2==0)
                      J1:  flag2=1
 // Critical            // Critical
 // Section             // Section
                (b)
```

Fig. 1: (a) shows Dekker's algorithm and (b) shows how an SCV can occur there. (c) Steps in detecting an SCV.

Detecting SCVs is crucial for the simplification of parallel programs. Maintaining SC is considered to be one of the major correctness criteria for parallel programs. Programmers can ensure SC semantics in any architecture by writing the programs in a data race free manner [2, 19]. However, the programs can have occasional data races (intentional or unintentional) and hence, SCVs (Section 2.2 discusses how data races and SCVs are related). The situation gets complicated when the memory model specifications of commercial processors from Intel and AMD do not even match the actual behavior of the machines [27]. Therefore, programmers may not be able to reason about SC behavior with those specifications. To make things worse, many work on semantics and software checking [26] that can potentially make parallel programming easier, would not be useful in the presence of SCVs. Thus, it is necessary to have a technique that can detect SCVs.

Significant research has been done to detect SC violations. One line of work [5, 32, 12] encode the program and memory model constraints as axioms and use a constraint solver to find violations of SC. There are some approaches [8, 7] that detect data races and find SCV cycles among them. Software based approaches described so far cannot be used during production runs. A recent study [14] has shown that many real world applications like Apache, MySQL, Mozilla, Gcc, Java, Cilk [1], Splash2 etc. have SCV bugs. Only 20% of the bugs are detected by software testing tools. The rest are discovered by programmers during their analysis of source code. As a result, a lot of SCV bugs remain hidden for a long time. Such findings warrant always-on hardware based solutions that can detect these bugs as soon as they occur.

Most of the hardware based approaches [3, 11, 31, 9, 19, 17] detect data races as proxies for SCVs. However, the number of data races can be two orders of magnitude higher than the number of SCVs [24]. Recent proposals like Volition [24], Vulcan [22], and Conflict Ordering (CO) [16] focus on detecting actual SCVs. They work by piggybacking memory reordering information with cache coherence protocol messages. They suffer from several limitations. *First,* they complicate coherence protocols significantly (e.g., Volition introduces *five* different network messages). *Second,* many of those proposals cannot work properly in the presence of branch mispredictions. *Third,* they require many hardware structures to be proportional to the number of processors and thus, are not suitable for higher processor count. *Finally,* existing hardware approaches provide very little debugging information. At most, they provide information about the last pair of memory accesses involved in an SCV. An SCV requires at least 4 memory accesses. Thus, the provided information is inadequate for a programmer.

This project aims to strike a balance between simplicity and effectiveness. We would like to propose a technique that can be used during production run without suffering from the previous shortcomings. Our proposed scheme, Dissector, works in two phases - an online phase to detect (potential) SCVs using a lightweight hardware and an offline post processing phase to filter out false alarms and extract detailed debugging information using a software. Dissector, targets TSO memory model for its widespread availability. In addition, it is streamlined for detecting 2 processor SCVs because of their sweeping majority [22, 14]. Dissector exploits the fact that TSO allows only one type of memory reordering - a load bypassing an earlier store. Therefore, an SCV can occur when the earlier bypassed store communicates with some remote load or store. Whenever a write miss (due to a store, S1) invalidates (step 1 in Fig. 1(c)) a line accessed by a load L0, the processor P0 responds (step 2) with a count of stores. The count includes all completed stores as well as any pending store that is earlier (according to the program) than L0. In a sense, the count expresses after how many stores, L0 appears to be ordered. Upon receiving the response, the processor P1 keeps tagging (step 3) the lines accessed by subsequent memory instructions with the count. When P0 sends an invalidation (due to a store, S0) to P1 (step 4), P0 piggybacks a count of its total completed stores. If this count is smaller than the tag stored with the invalidated line, S0 must be one of P0's pending stores that initially got bypassed by L0. Hence, an SCV is reported by the Dissector hardware. The report consists of two instructions - S0 and the memory instruction, A1 that accessed the invalidated line in step 4. However, a 2-processor SCV requires 4 instructions - S0, L0, S1, and A1. In order to determine the other two instructions, Dissector keeps logging every communicating pair like L0 and S1, where L0 is a bypassing load that gets invalidated by S1 (before the prior stores of L0 are completed). Let us denote (L0, S1) as the First Pair (FP) and (S0, A1) as the Second Pair (SP). The post processing software takes the report of FPs and SPs and enumerates over their possible combinations. For each combination, it profiles some memory accesses and applies Shasha-Snir's SCV detection algorithm [28] to either confirm a true SCV or prune a false alarm. Since Dissector is a two-phased detection scheme, we envision its usage model to either (ii) reactive (default mode) where the report is processed only after a failure (crash, incorrect result etc.) occurs or (ii) proactive where the report of potential SCVs are processed immediately after the execution.

Dissector hardware relies solely on messages generated by cache coherence protocols. It does not introduce any new messages. It does not alter the behavior of coherence protocols either. It only piggybacks few extra bytes with existing coherence messages. Dissector requires a small amount of hardware structures per processor. The hardware requirement does not change with processor count. Dissector hardware works seamlessly with branch mispredictions. Dissector, with the help of its post processing phase, prunes false positives and provides detail information (i.e., all instruction and memory addresses) about true SCVs. Dissector is unconcerned about compiler induced SCVs. The paper presents a detail design of Dissector. We evaluated it in a multiprocessor system using a cycle accurate simulator [25] and Pin [18]. We experimented with different concurrent algorithms, bug kernels, SPLASH2, and PARSEC applications. Our results show that even with a simple non-intrusive design, Dissector has a better SCV detection ability than a prior state-of-the-art technique. For a 4-processor system, it incurs a negligible execution and network overhead of 0.02% and 2.78% respectively. It requires only 3.5KB hardware per processor.

This paper is organized as follows. Section 2 gives a brief overview of background; Section 3 describes Dissector design; Section 4 explains implementation issues; Section 5 presents experimental results; Section 6 discusses related work; and finally, Section 7 concludes the work.

2 Background

2.1 TSO Memory Model

A TSO machine has a write buffer with each processor. When a store reaches the head of the Reorder Buffer (ROB), it retires into the write buffer. From there, the stores are performed in order. A store is *completed* when the local cache receives all invalidation acknowledgements for the write. When a store is completed, it is removed from the write buffer. Whenever a load reaches the head of the ROB and the data is returned from the local cache, it is allowed to retire even if the write buffer contains some earlier stores. This process essentially lets a load to bypass earlier stores in TSO. A load is said to *complete* when it retires from the ROB.

2.2 Patterns for an SCV

Shasha and Snir [28] show what leads to an SCV: overlapping data races that cause dependences to form a happened-before cycle at runtime. Recall that a data race occurs when two (or more) threads/processors access the same location without an intervening synchronization and at least one is writing. Fig. 2(a) shows the required program pattern for two processors (where each variable is written at least once) and Fig. 2(b)-(d) show the required order of the dependences for SCVs in TSO. The dependences are Write-After-Read or Write-After-Write dependences. Each arrow is shown from the earlier access to the later one. So, we refer to the earlier access as the *Source* and the later access as the *Destination* of the dependence.

If at least one of the dependences occur in the opposite direction or any other pattern appears at runtime, no cycle can form and hence, no SCV occurs. Note that for the pattern in Fig. 2(b), each dependence occurs between a store and a remote load whereas for the patterns in Fig. 2(c) and (d), the dependences occur between a store and a remote load/store. Thus, in TSO, an SCV can occur when a store depends on a remote load/store.

3 Dissector: A Hardware Software Co-Designed Approach

Dissector consists of a lightweight hardware to detect SCVs and a post processing software to prune false alarms later. We will start by explaining the overall approach of

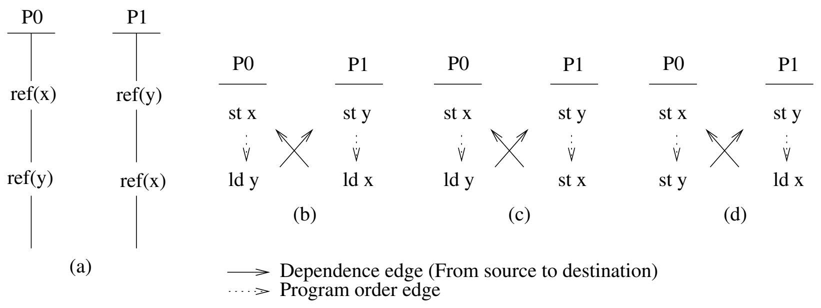


Fig. 2: Understanding SCVs in TSO.

Dissector hardware assuming a two processor system with a single word cache line in section 3.2. Section 3.4 & 3.5 extend the design to handle multiword cache line and more than two processors respectively. Section 3.6 handles all the subtleties of cache coherence protocol. Finally, Section 3.7 describes the post processing analysis. Keep in mind that Dissector is designed to detect two processor SCV cycles.

3.1 Definitions

We start by defining some terms that will be used throughout this section. (i) Completed Store Counter (CSC) is a per processor counter to keep track of the stores that the processor completed. (ii) If a processor completes a load while some earlier stores are pending, Violating Store Point (VSP) denotes the number of total completed stores (from the same processor) after which the load appears to be ordered. If the count of earlier pending stores is denoted by PS, then VSP is essentially the sum of CSC and PS i.e., $VSP = CSC + PS$. (iii) Each processor assigns a Serial Number (SN) to a memory reference instruction during the issue stage. SN is a scalar quantity that starts from 1 for the first memory reference instruction of a processor and keeps incrementing for subsequent memory reference instructions. SN is used to determine program order among memory reference instructions. (iv) For a multiword cache line, we keep 1 bit per word to indicate whether any access to that word can potentially cause an SCV. The bit is referred to as Unsafe (U) bit. (v) Finally, a two processor SCV cycle consists of two dependences (Section 2.2). The dependence that occurs first is referred to as the First Pair (FP) and the other one is referred to as the Second Pair (SP).

3.2 Basic Operation of Dissector Hardware

Let us assume that a processor, say P_0 completes a load L and the line accessed by the load gets invalidated by a remote store. P_0 responds with VSP. Recall that $VSP = CSC + PS$ where PS is the number of pending stores that are earlier than L. After P_0 completes a total of VSP stores, the load appears to be ordered and no longer causes any SCV. If the invalidated line of P_0 was last accessed by a store S (instead of the load L), the store should be already ordered in TSO i.e., $PS = 0$. So, P_0 responds with $VSP = CSC$.

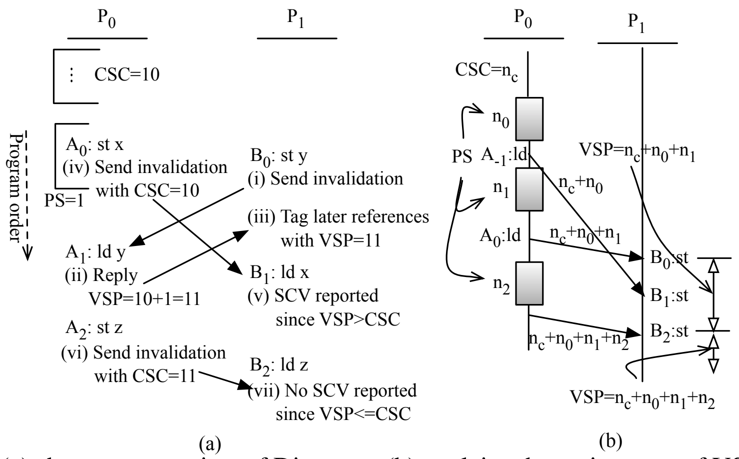


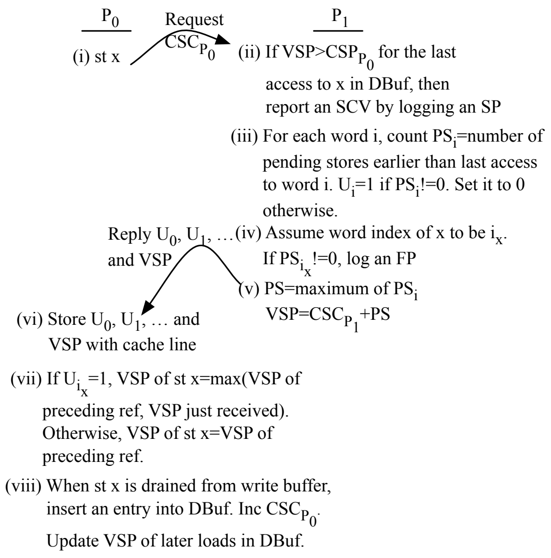
Fig. 3: (a) shows an overview of Dissector. (b) explains the assignment of VSP.

Let us consider the example in Fig. 3(a) where the load A_1 bypasses the pending store A_0 in processor P_0. Assume that P_0 has completed $CSC = 10$ stores so far. When P_0 receives an invalidation due to the store B_0, A_0 is still pending (i.e.,, $PS = 1$). So, P_0 replies with $VSP = CSC + PS = 11$. Thus, P_0 is letting P_1 know that the load A_1 will appear to be ordered when P_0 completes a total of 11 stores. P_1 starts tagging all later (issued) references (e.g., B_1 & B_2) with $VSP = 11$. When A_0 generates an invalidation request, P_0 sends CSC (which is still 10) along with the request. The load of the invalidated line, B_1, is tagged with $VSP = 11$. Since CSC has not reached VSP yet, P_0 has not completed all necessary stores to make A_1 appear to be ordered yet. This implies that the invalidation is coming from the pending store A_0 which was bypassed by A_1. Thus, the reordering of A_0 and A_1 gets exposed to P_1 and an SCV is detected. When A_0 completes, CSC of P_0 becomes 11. Now consider the store A_2 which is younger than A_1 and hence, is not reordered with A_1. When A_2 causes an invalidation request, P_0 sends CSC=11 with it. The load of the invalidated line, B_2 has VSP=11. Since CSC has reached VSP, P_0 has completed all the stores necessary to make A_1 appear to be ordered. So, no SCV is detected.

Note that the dependence $B_0 \rightarrow A_1$ starts the happened-before cycle. Therefore, FP is the instruction pair (B_0, A_1). The dependence $A_0 \rightarrow B_1$ finishes the happened-before cycle and therefore, SP is the instruction pair (A_0, B_1). When A_0 causes an invalidation and an SCV is detected with B_1, Dissector hardware logs the instruction address of A_0 and B_1 as SP in a memory mapped file. We assume that instruction address of A_0 is piggybacked with the invalidation message. To capture FP, Dissector hardware finds every instance where the line accessed by a bypassing load (e.g., A_1) is invalidated due to a store (e.g., B_0) and logs the instruction address of A_0 and B_1 as FP. This will cause dependences other than $B_0 \rightarrow A_1$ to be logged as FPs as well. Those are filtered by the post processing software.

3.3 How VSP is Assigned?

Consider Fig. 3(b). Assume that P_0 has already completed a total of $CSC = n_c$ stores. Processor P_0 has some pending stores. The topmost box indicates a portion of execution



P0
P1
Request CSC_P0
(i) st x
(ii) If VSP>CSP_P0 for the last access to x in DBuf, then report an SCV by logging an SP
(iii) For each word i, count PS_i=number of pending stores earlier than last access to word i. U_i=1 if PS_i!=0. Set it to 0 otherwise.
(iv) Assume word index of x to be i_x. If PS_ix!=0, log an FP
(v) PS=maximum of PS_i VSP=CSC_P1+PS
Reply U0, U1, … and VSP
(vi) Store U0, U1, … and VSP with cache line
(vii) If U_ix=1, VSP of st x=max(VSP of preceding ref, VSP just received). Otherwise, VSP of st x=VSP of preceding ref.
(viii) When st x is drained from write buffer, insert an entry into DBuf. Inc CSC_P0. Update VSP of later loads in DBuf.

Fig. 4: SCV detection using CSC and VSP.

where P_0 has n_0 pending stores. Subsequent boxes represent more execution portions where P_0 has n_1 and n_2 pending stores respectively. Processor P_1 sends a request to P_0 due to a store B_0. This creates a (write-after-read) dependence $A_0 \rightarrow B_0$, where A_0 is a load from P_0. As in Section 2.2, the dependence arrow is shown from the earlier access to the later access. P_0 responds with $n_c + n_0 + n_1$ as VSP. Any of the $n_0 + n_1$ pending stores from P_0 can cause an SCV with B_0 or later reference instructions. In other words, those pending stores can expose the reordering of A_0. Therefore, any memory reference instruction from B_0 to B_1 is tagged with $n_c + n_0 + n_1$ as VSP. For store B_1 in P_1, P_0 responds with $n_c + n_0$ as VSP (due to the dependence $A_{-1} \rightarrow B_1$). Note that even if some (say, $n_{0'}$) of the n_0 pending stores complete by the time B_1 causes an invalidation request, CSC will be increased to $n_c + n_{0'}$ while PS will be decreased to $n_0 - n_{0'}$. At the end, VSP returned by P_0 will still be $n_c + n_0$. Therefore, for the sake of simplicity, we can assume that P_0 does not complete any of its pending stores for the rest of the discussion. B_1 causes the receipt of $n_c + n_0$ as VSP. Thus, the dependence $A_{-1} \rightarrow B_1$ allows B_1 and later memory reference instructions of P_1 to have an SCV with any of the n_0 pending stores from P_0. On the other hand, the dependence $A_0 \rightarrow B_0$ allows B_1 and later reference instructions to have an SCV with any of the n_0 as well as n_1 pending stores from P_0. Therefore, B_1 and later reference instructions (up to B_2) are tagged with $n_c + n_0 + n_1$ as VSP. In other words, VSP of a memory reference instruction is set to the larger of the two - VSP of the preceding (in program order) memory reference instruction and the VSP received, if any, from another processor. Note that VSP is received only for a store. Therefore, a load simply inherits its VSP from the preceding reference instruction. Finally, store B_2 receives $n_c + n_0 + n_1 + n_2$ as VSP from P_0. This is larger than the VSP of the preceding reference instruction (which is $n_c + n_0 + n_1$). Hence, B_2 and later memory reference instructions have $n_c + n_0 + n_1 + n_2$ as VSP. A curious reader might wonder what happens if P_0 completes all of its pending stores before receiving the invalidation of B_0. In that case, B_0 and later reference instructions will have $n_c + n_0 + n_1 + n_2$ as

VSP. Although this is an over-estimated value, invalidations of future stores from P_0 will have at least $n_c + n_0 + n_1 + n_2$ as CSC and hence, no false positives will occur.

Note that VSP of a memory reference instruction is used when a remote store has a dependence with it. A memory reference instruction has to complete before a remote store can depend on it. Therefore, a processor assigns VSP to a memory reference instruction when it completes. Thus, misspeculated loads are automatically discarded by Dissector hardware. When a memory reference instruction completes, the ROB (or, write buffer) no longer holds that reference. Therefore, each processor uses a buffer, called *DBuf*. DBuf keeps the reference instructions according to the order of issue (i.e., based on SN). When a memory reference instruction completes, SN and VSP are kept along with its memory and instruction address in DBuf. We only need to keep the last reference instruction to a particular address for SCV detection. Therefore, any new entry in DBuf can cause removal of earlier entries (i.e.,, the ones with smaller SN) with the same memory address. This buffer is checked in parallel with the local cache when an invalidation request arrives.

In TSO, when a store completes, some of the later loads from the same processor might already have completed. Therefore, when the store completes and VSP is assigned to it, a processor needs to check later loads (i.e.,, the ones with larger SN) and possibly update their VSPs. Recall that when a load completes, it inherits its VSP from the preceding memory reference instruction. The preceding reference instruction can be a pending store with no VSP assigned yet. In that case, the processor keeps inspecting the reference instructions in decreasing SN order until it finds one with an assigned VSP. The load simply inherits that VSP. Eventually, as the pending stores complete, the load gets its VSP updated.

3.4 Handling Multiword Cache Line

The algorithm described so far, works fine for a single word cache line system because any store that creates an interprocessor dependence causes a cache coherence message and the processors can piggyback CSC and VSP with those messages. For a multiword cache line system, not all stores that create interprocessor dependences cause cache coherence messages. Therefore, anytime a store generates a cache coherence message, the processors need to piggyback information not only for the requested word but also for other words in the same line. Assume that each cache line contains W words.

The straightforward way to extend the single word cache line algorithm is to send separate VSP for each word in the same cache line. Communication and storage of such VSPs would cause significant overhead. Therefore, a processor sends only 1 VSP for the entire cache line and associates 1 U bit with each word to indicate whether any access to that word can be potentially involved with an SCV. The algorithm is shown in Fig. 4. When processor P_0 sends a request due *st x*, it sends CSC_{P_0}. P_1 finds the last reference instruction to x in its DBuf and checks if the associated VSP is larger than CSC_{P_0}. If so, P_1 reports an SCV by logging the relevant instructions as an SP. After checking for an SCV, for each word i, for $0 \leq i < W$, P_1 counts the number of pending stores PS_i earlier than the last access to that word. If PS_i is not 0, there are some earlier pending stores that can cause an SCV with a remote access to the word.

Thus, the remote access could be unsafe and so, unsafe bit U_i associated with the word is set. If there are some pending stores before the last access to word x (i.e.,, $PS_{i_x} \neq 0$), the dependence is logged as an FP. P_1 summarizes all PS_i by taking the maximum, denoted by PS. Thus, an access from P_1 to any word of the cache line bypasses at most PS pending stores. So, PS is a conservative estimate of pending stores and can lead to false positives. VSP is calculated by adding PS and CSC_{P_1}. P_1 sends all U_i bits and VSP with the reply message. After receiving these, P_0 stores U_i bits and VSP with the cache line so that they can be used in future. If the unsafe bit associated with word x (i.e., U_{i_x}) is set, P_0 sets VSP of *st x* as the VSP of the preceding memory reference instruction or the VSP just received, whichever is the larger. If, however, U_{i_x} is cleared, there are no pending stores in P_1 (before its last access to x) that can cause an SCV. Hence, *st x* copies its VSP from the preceding reference instruction. When the store is drained from P_0's write buffer, an entry is inserted into DBuf, CSC_{P_0} is incremented and VSPs of later loads are updated as usual.

3.5 Handling More Processors

Assume that the system has N processors for $N > 2$. Here, when a processor P_i, for $0 \leq i < N$, sends an invalidation due to a store, more than one processor can reply. The reply from processor P_j, for $0 \leq j < N$ and $j \neq i$, contains VSP_{P_j} and unsafe bits U_{jl}, for $0 \leq l < W$. P_i combines the replies. If a reply has all unsafe bits cleared, the corresponding processor has all of its last accesses to the line appear to be ordered. Hence, the reply is not considered during the combination process. From the remaining replies, unsafe bits are combined by taking the logical OR of the corresponding bits. So, if a word is marked unsafe at least in one reply, it is also marked unsafe after the combination. Thus, the resultant reply contains U_m bits, for $0 \leq m < W$, where $U_m = OR(U_{0m}, ..., U_{(N-1)m})$. VSPs are conservatively combined by taking the maximum from the remaining replies. Such merging can lead to false positives. Thus, the algorithm in Fig. 4 remains the same except that P_0 needs to combine the replies before applying steps (vi)-(viii).

3.6 Issues with Cache Coherence Protocol

Let us consider a bus based snoopy system. Section 4 explains a directory based scheme. Without loss of generality, let us assume an MSI protocol. We will discuss all cases – store miss/hit and load miss/hit. When a processor suffers a write miss due to a store, it broadcasts an invalidation. Every processor snoops on the bus and responds. All the steps mentioned in Fig. 4 are applied. When a store causes a write hit, the associated cache line contains unsafe bits and VSP which are used to calculate VSP of the store and possibly update VSPs of later loads in DBuf. However, a write hit can lead to both false positives and negatives. A write hit implies that the cache line is in modified state. Hence, no other processor has accessed it since the completion of the store that originally brought the line in modified state. Therefore, there is no new pending store from other processors that precedes those processors' last access to the line. Hence, the associated VSP still correctly specifies the stores (from those other processors) that can cause an SCV with an access to this line. So, VSP associated with the modified line is still accurate. The unsafe bits, however, may stay unsafe for longer. This is due to the fact that the pending stores might have completed. Moreover, instead of a write hit if the store could cause a write miss, other processors would have received an invalidation request, checked for an SCV with their last access to the requested word, and (sometimes) logged an FP. Such checking and logging cannot be done when a write hit occurs to a previously unaccessed word. Thus, when a write hit happens for a previously unaccessed word, the requested processor can end up using overestimated information (due to unsafe bits) which can cause false positives, other processors can lose a chance to detect an SCV resulting in false negatives, and some FPs may not be logged. False positives will be pruned by the post processing step. Missing SCVs (i.e.,, false negatives) will eventually be detected since we envision the hardware to be active in every execution even during the production run. Missing of some FPs are discussed in Section 3.7.

A load always inherits its VSP from the preceding (or even earlier) memory reference instruction. Therefore, a cache line that is brought due to a read miss has its unsafe bits and VSP assigned to the initialized values (i.e., all 0s). Any future write miss on the same line brings up-to-date VSP and unsafe bits. A read (hit/miss) simply causes the associated load to get its VSP from the preceding (or even earlier) instruction.

3.7 Postprocessing by Dissector Software

The goal of the postprocessing software is to filter out false SCVs and with the help of FPs and SPs, provide detail information for true SCVs. Recall that an FP is a dependence between a store and a load that bypasses some stores in another processor. An SP is a dependence between a store and a load/store in another processor where an SCV is detected. A 2-processor SCV consists of an FP and an SP. Consider the SCV in Fig. 3(a) where FP is the dependence between B_0 and A_1, and SP is the dependence between A_0 and B_1. Dissector hardware logs a set of FPs, SPs along with the id processors involved . An SP can be associated with any one of the FPs to cause an SCV. Therefore, Dissector software checks all possible combinations of FPs and SPs. Let us consider a combination where FP is between a store F_s and a load F_l, and SP is between a store S_s and a load/store S_{ls}. According to Shasha and Snir [28], this combination can cause an SCV if (i) there are data races between S_s and S_{ls} as well as F_s and F_l, (ii) S_s and S_{ls} access a location different than F_s and F_l, (iii) in the program, S_s is earlier than F_l in the same thread and F_s is earlier than S_{ls} in the same thread, and (iv) there is no fence between S_s and F_l (Fig. 5). One might wonder why we did not consider the case where there is a fence between S_s and F_l but no fence between F_s and S_{ls} (when S_{ls} is a load). Such a scenario can cause an SCV due to the reordering of S_{ls} and F_s. Therefore, (S_s, S_{ls}) would be logged as an FP instead of an SP and vice versa. Thus, without the loss of generality, we consider the absence of a fence only between S_s and F_l as the required constraint.

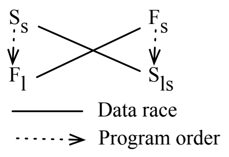


Fig. 5: Required constraints for an SCV.

To check the constraints for a combination (S_s, S_{ls}, F_s, F_l), the program is run with a profiler using the same inputs as the original run. The profiler profiles S_s, S_{ls}, F_s, and F_l instructions. It also profiles any other instruction that accesses the same locations as these instructions. For each of these instructions, it records the instruction and memory address and the id of the executing thread. The profiler captures the order of execution of different memory access instructions from the same thread. The profiler also captures any fence and synchronization operation executed. The output of the profiler is a file that contains all these information. A happened-before race detection [23] algorithm is applied to the contents of the file. If a pair of memory accesses to the same location do not have any happened-before relation and at least one of them is a store, the pair is marked as a racing pair. Any instance of S_s, S_{ls}, F_s, or F_l that is not involved in a data race is discarded from further considerations. Algorithm 1 is then applied. It checks one thread at a time. It finds every instance of S_s and F_l in the same thread where S_s is earlier than F_l in the program and is not intervened by a fence or a local store to the same location as F_l. If such an instance exists, it finds every instance of F_s and S_{ls} that races with F_l and S_s respectively. If, at least once, F_s executes before S_{ls} by the same thread, then we identify a scenario where the combination (S_s, S_{ls}, F_s, F_l) can cause an SCV. If such a scenario is not found, the combination is filtered out as a false positive. In any case, the software then applies the same algorithm for other combinations of FPs and SPs.

Algorithm 1 Processing a combination (S_s, S_{ls}, F_s, F_l)

```
for each thread $t$ do
    for each $S_s$ in $t$ do
        for each $F_l$ that is later than $S_s$ in $t$, accesses a location different than $S_s$, and is not intervened by a fence or a
        store to the same location as $F_l$ in $t$ do
            for each $S_{ls}$ that (data) races with $S_s$ do
                for each $F_s$ that (data) races with $F_l$ do
                    Check if $F_s$ and $S_{ls}$ are from the same thread $r$ such that $r \neq t$ and $F_s$ is earlier than $S_{ls}$.
                    If so, confirm an SCV and break.
                end for
            end for
        end for
    end for
end for
```

Note that our software phase relies on the presence of data races to confirm an SCV. It is possible that the required data races might not occur when we run the program with the profiler. To remedy this, we inject random delay during profiling and run the profiler several (e.g., 20) times. Since we are focusing mostly on 4 instructions at a time, it is even possible to consider tools like CHESS [21] to generate all possible interleavings. Finally, a write hit can cause the missing of an FP. To remedy this, we can record FPs found during different executions and use them all to generate different combinations with a set of SPs.

4 Implementation Issues

Additional Hardware Structures Each processor is equipped with DBuf. DBuf holds a memory reference instruction and associated information after it completes. Since an entry is allocated after the completion of a reference instruction, the allocation process is outside the critical path of the pipeline. DBuf is implemented as a circular ordered list as shown in Fig. 6. The entries are ordered according to SN. The entry for the oldest reference is pointed to by *tail* and the newest reference is pointed to by *head*. The list grows up to a maximum size. When it reaches that size and a new memory reference instruction needs to be inserted, the oldest reference instruction is removed from the tail and the new reference instruction is added to the head. When a memory reference instruction is completed and the (word) address it accessed is already present with some older entry in the list, the older entry is removed and the newer one is inserted. DBuf might need to be accessed using a memory (word) address. To facilitate this process, a hash table is associated with the list (Fig. 6). Each entry in the hash table contains memory address and a pointer to an entry in the list that accessed the same address.

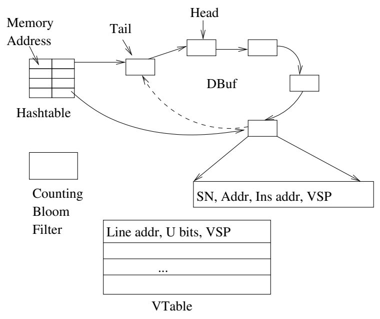


Fig. 6: Hardware structures required for Dissector.

So far we assumed that unsafe bits and VSP of a cache line is stored in the cache with the line itself. This can lead to significant overhead. Therefore, we keep unsafe bits and VSP in a per-processor small cache like structure, called VSP Table (VTable) as shown in Fig. 6. This is similar to Volition [24]. When a write miss brings unsafe bits and VSP, an entry containing them is inserted into VTable. The entry does not store actual data. When a line is invalidated or evicted from the processor's cache and VTable contains an associated entry, the entry is removed from VTable.

Handling Cache Line Evictions When a line is evicted from the cache, some of the words of that line might still be present in the associated DBuf. We propose to use a Counting Bloom Filter (CBF) [4] that hashes (word) addresses of all entries of DBuf (similar to prior schemes [22, 24]). When there is an invalidation request in the bus, the CBF is checked to determine if the requested word may be present in DBuf. If so, the entry corresponding to the word is searched in DBuf and processed.

Handling a Directory Based Scheme Dissector requires that a processor receives an invalidation request for any address in DBuf. If the corresponding line is present in the cache, the directory will send an invalidation request. However, if the line is not present, no invalidation will be sent. This issue is partially addressed if cache lines are evicted silently. Next store to the same line will cause the directory to send an invalidation request to processors whose cache previously contained that line. After this point, the directory will have updated information about the sharers and no more invalidations will be sent to those processors. However, those processors might still have some words of the evicted line in DBuf. Although it is possible to force the directory to send future invalidations to those processors with the help of some CBFs that keep track of the addresses in DBuf, such modifications will complicate the directory protocol. So, we choose not to change the directory protocol and accept few more false negatives.

Handing a Race Condition Consider Fig. 3(a) where A_1 and B_1 are completed. Now, A_0 and B_0 try to complete simultaneously. As a result, before P_0's response arrives and changes VSP of B_1, P_1 checks for SCV at B_1 and detects no SCV. Similarly, before P_1's response has a chance to update VSP of A_1, P_0 checks for SCV at A_1 and detects no SCV. To prevent this race condition, whenever a processor handles an incoming invalidation request while one of its pending stores is in progress (i.e., already sent out invalidation request), the processor serializes the processing of requests according to some pre-defined order based on processor id. In the previous example, lets assume that the order is P_0 and then, P_1. In that case, P_0 will not process P_1's invalidation request until P_0 receives the response for its ongoing store A_0. The response will update VSP of A_1 and then P_0 handles P_1's request and detects an SCV. On the other side, P_1 does not wait for P_0's response due to B_0 and processes the incoming request due to A_0. P_1 does not detect any SCV and responds back to P_0. Eventually when P_0's response arrives, it updates VSP of B_1. The same principle can be applied to any number of processors.

Wrap-Around of VSPs, CSCs and SNs When wrap-around occurs, two numbers that should be comparable become very far apart. Therefore, it is possible to detect this event by looking at few higher order bits. If they are completely opposite, Dissector hardware can realize that the smaller number is supposed to be higher than the other one.

5 Evaluation

Experimental Setup We model Dissector hardware using a cycle accurate execution driven simulator [25]. We simulate a chip multiprocessor with private L1 caches and a shared L2 cache. Table 1 shows the architectural parameters. When there is a choice, the values in bold are the default ones. We use PIN [18] to write the profiler.

We use three sets of benchmarks for evaluation (Table 2). The first set has implementations of concurrent data structures and mutual exclusion algorithms that have potential SCVs [5, 6]. The second set has some reported SCV bugs from open source programs and libraries (e.g., MySQL, Gcc, Cilk). Finally, we use eight applications from SPLASH-2 and two applications from Parsec.

SCV Detection To measure Dissector's SCV detection ability, we run each application multiple times - the smaller ones 100 times and the larger ones (i.e. SPLASH2 & Parsec) 5 times. In each run, we force different interleavings by introducing some randomness. For each application, we collect, over all the runs, the number of unique SPs and FPs observed. The post processing software takes the report of FPs and SPs and enumerates

Architecture	Chip multiprocessor with **4**, 8 or 16 cores.
Core pipeline	Out-of-order; 3.0GHz; 2-issue/2-retire.
ROB size	128 entries.
Write buffer size	16 entries.
Private L1 cache	32KB WB, 4-way associative, 6-cycle rt.
Shared L2 cache	1MB WB, 8-way associative, 12-cycle rt.
Cache line size	**32B** or 64B.
Coherence	Snoopy MSI protocol; 3.0GHz 32B-wide bus.
Consistency	TSO
Memory	300 cycle rt.
Dissector Parameters	DBuf: 32, **128** or 1024 entries. SN, VSP, CSC: 4B each. VTable: 32, 64 or **128** entries. CBF: 128B with 2 bit counters, H3 hash.

Table 1: Multicore architecture evaluated.

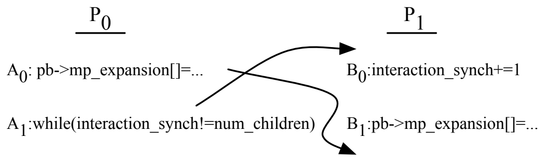
Set	Program	Description
Conc. Algo.	dekker	Algo. mutual exclusion.
	snark	Non-blocking double-end. queue.
	msn	Non-blocking queue.
	harris	Non-blocking set.
	lazylist	List-based concurrent set.
	peterson	Algo. for mutual exclusion.
Bug kernels	pthread_cancel from glibc	Unwind code after canceling thread needs a fence [22].
	crypt_util from glibc	Small table initialization code needs a fence [22].
	init from MySQL	Available charsets initialization code needs a fence [14].
	Cilk_unlock from cilk	Cilk_unlock needs full fence instead of store-store fence [8].
Full Apps	SPLASH-2	8 programs form SPLASH-2.
	Parsec	2 programs form Parsec.

Table 2: Applications analyzed.

over their all possible combinations. For each combination, it either confirms a true SCV or prunes a false alarm. We compare our scheme against an existing hardware based SCV detector, Vulcan [22]. The comparison remains the same even if we consider Volition [24] that is tuned to detect 2 processor cycle (*Volition**). Fig. 7(a) shows the results and comparisons for different applications.

Codes	FP	SP	Total comb.	Filtered comb.	True SCV	Vulcan/ Volition*
harris	2	1	2	2	0	0
lazylist	0	1	0	0	0	0
msn	3	1	3	3	0	0
snark	3	2	6	6	0	0
crypt_util	2	2	4	2	2	2
pthread_can.	4	4	16	13	**3**	2
dekker	2	1	2	2	0	0
peterson	3	3	9	5	**4**	3
init	2	4	8	7	1	1
Cilk_unlock	2	0	0	0	0	0
fft	5	1	5	5	0	0
radix	8	5	40	40	0	0
lu	2	0	0	0	0	0
ocean	121	5	605	605	0	0
water-ns	9	11	99	99	0	0
water-sp	9	7	63	63	0	0
barnes	29	15	435	435	0	0
fmm	35	13	455	452	3	3
swaptions	10	2	20	20	0	0
stream.	1	8	8	8	0	0
Total	252	86	1780	1767	13	11

(a) SCVs found in different applications


P_0
P_1
A_0: pb->mp_expansion[]=...
B_0:interaction_synch+=1
A_1:while(interaction_synch!=num_children)
B_1:pb->mp_expansion[]=...

(b) New SCV found in fmm

Fig. 7: Detected SCVs

Fig. 7(a) shows that Dissector hardware logs a total of 252 FPs and 86 SPs. The post processing software enumerates over 1780 combinations. For each combination, it collects a number of profiles (up to 20). It filters 1767 combinations as false alarms and reports detail information (i.e. instruction and memory addresses of all accesses) for the rest (i.e., 13) of the SCVs. Except for *pthread_cancel* and *peterson*, both Dissector and Vulcan/Volition* detect equal number of the SCVs. Dissector detects more SCVs in those programs. This is due to the fact that Vulcan/Volition* identifies an SCV only by the last pair of instructions (i.e., SP). Therefore, multiple different SCVs might be reported as a single one. Dissector, on the other hand, is able to distinguish and report them as separate SCVs. Note that even with a simpler and smaller hardware, Dissector does not have any false negatives.

We like to understand whether profiles of data races can be used in conjunction with a software based scheme such as Relaxer [7] to find out SCVs. We used *fmm* as an example. We found 15 data races using Intel Inspector [13]. The profile contained 24.3 million accesses. It was too much to be used with Relaxer. So, profile based software only schemes are not suitable especially for large applications.

We found a *previously unreported* SCV in *fmm*. It was detected by both Dissector and Vulcan/Volition*. In *fmm*, different threads can process *boxes* in opposite order. This can lead to an interleaving shown in Fig. 7(b). Here, processor P_0 reads *interaction_synch* in A_1 before modifying *mp_expansion* in A_0. Another processor P_1 modifies *interaction_synch* in B_0 and then modifies *mp_expansion* in B_1. Although no reordering is possible between B_0 and B_1 in TSO, the reordering of A_0 and A_1 causes an SCV. Note that *interaction_synch* is declared as *volatile* in code. However, its read in A_1 can still bypass A_0 and cause an SCV. To fix this bug, *interaction_synch* needs to be declared as *atomic* in C/C++.

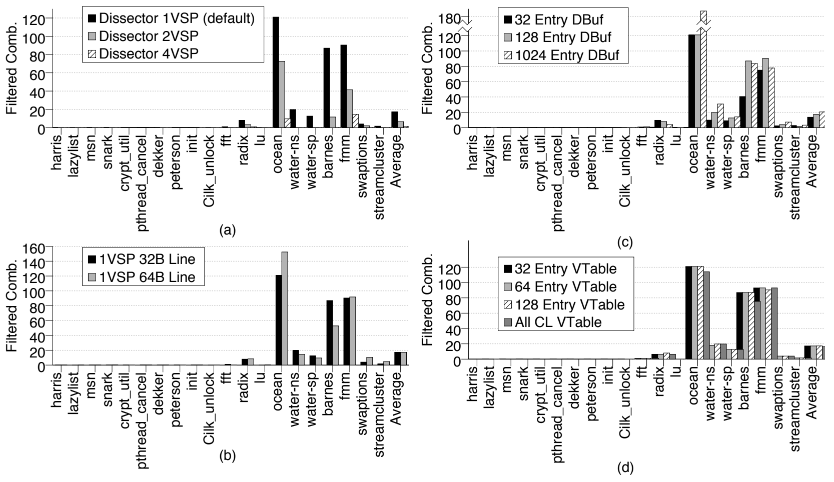
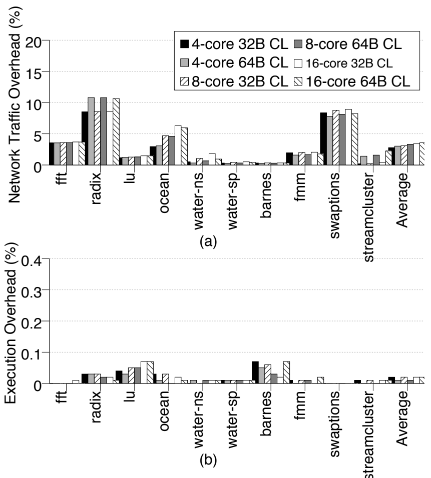


Fig. 8: Sensitivity to (a) number of VSP, (b) cache line size, (c) size of DBuf, and (d) size of VTable.

Sensitivity Analysis We evaluate three choices - 1VSP (default), 2VSP, and 4VSP per cache line. We compare each with Vulcan/Volition*. Dissector does not have any false negative in any case. We count the average number of false SCVs filtered by the postprocessing software (Fig. 8(a)). The average is calculated for each execution. On average, for each application, Dissector filters 17.29 combinations in the default version. However, 2VSP and 4VSP filter 6.55 and 1.24 combinations per application respectively. Recall that the filtering is done offline by the post processing software and the default choice is to do it only after a failure execution. We opt for 1VSP as our default design.

We experiment with 2 cache line sizes - 32 and 64 Byte. Dissector does not have any false negative for 32 Byte line but 1 false negative in *peterson* program for 64 Byte line. Fig. 8(b) shows the average number of combinations filtered for each execution. For 32 Byte line, the average is 17.29 per application whereas for 64 byte line, the number is 17.27. Dissector has two structure DBuf and VTable. In order to assess the impact of DBuf, we keep the size of VTable to be 128 and change the size of DBuf to be 32, 128, and 1024. For 32 entry DBuf, Dissector has 1 false negative in *init* program. For larger sizes, Dissector does not have any false negative. The filtered combinations are shown in Fig. 8(c). On average, for each application, the post processing software filters 13.59, 17.29, and 20.45 combinations per execution for 32, 64, and 128 entry DBuf respectively. We keep the size of DBuf to be 128 and change the size of VTable to be 32, 64, 128. We also simulate a case where VSP is stored with each cache line (*All CL*). There are no false negatives in any case. On average, for each application, the number of filtered combinations per application per execution are 17.25, 17.25, 17.29, and 16.41 for 32, 64, 128 entry VTable and All CL configuration respectively (Fig. 8(d)).



Criteria	Dissector	Vulcan	Volition
Add. message	**None**	1 type	5 type
Coh. protocol	**Unmodified**	Modified	Modified
Source of false positive	**None**	False sharing (bit level), branch mispred.	False sharing (bit level), branch mispred.
Source of false negative	Limited hw, some write hit	Limited hw	Limited hw
Per core hw for 4, 8, 16 core	3.5, 3.5, 3.5 KB	9, 17, 33 KB	6, 6, 6 KB
Scope	2 proc SCV	2 proc SCV	2+ proc SCV
Debugging info.	**All inst. & mem. addr.**	Last 2 inst addr.	Last 2 inst. addr.

(c) Comparison of Dissector, Vulcan and Volition.

Fig. 9: (a) shows network traffic overhead, (b) shows execution overhead, and (c) shows comparison.

Network Traffic & Execution Overhead We calculate execution and network traffic overhead. For overhead calculation, we use only large applications (i.e. SPLASH2 and Parsec). The overheads are calculated with respect to a baseline TSO machine. Fig. 9(a) shows the network overhead due to the piggybacking of VSP and unsafe bits with write misses. On average, the overhead for a 4-core default system is ($\approx$) 2.8%. It increases by less than 1% for higher processor count. The piggybacked traffic causes an average slowdown of ($\approx$) 0.02% for a 4-core system (Fig. 9(b)). This overhead remains virtually the same for more processors. The post processing phase requires 0.007s, 0.005s, 0.006s, 0.007s and 2456s to confirm true SCVs in *crypt*, *init*, *peterson*, *pthread_cancel* and *fmm* respectively. To discard a false alarm, it takes 5 hours in the worst case. This happens for *fmm*. Recall that post processing is done offline and the default choice is to do it only after a failure execution.

6 Related Work

The table in Fig. 9(c) shows a comparison between Dissector and the closest related work Vulcan & Volition. Besides them, majority of the existing work to detect SCVs focus on data races. Specifically, one line of work detects incoming coherence messages on data that has local outstanding loads or stores. This work started with Gharachorloo and Gibbons [9] and now includes many aggressive speculative designs (e.g., [3, 11, 31]). Another line of work detects a conflict between two concurrent synchronization-free regions. This includes DRFx [19] and Conflict Exceptions [17]. In general, all of these works look for a data race with two accesses that occur within a short time. Dissector, on the other hand, detects SCV cycles, not just data races. There are many proposals to implement SC. Most proposals to implement SC fall under two categories - in-window speculation [10] and post-retirement speculation [3, 11, 31]. At the high level, these proposals allow some accesses that would have been stalled in SC, to proceed speculatively. In case, there is a possibility of an SCV, the speculative accesses are squashed and retried. Some recent work [16, 29] has been proposed that does not rely on speculation. Conflict Ordering [16] ensures SC by allowing an access to bypass a prior pending access unspeculatively. Singh et al. [29] proposed to implement SC by enforcing order only among shared accesses. Marino et al. [20] used the same principle to implement an SC preserving compiler. Dissector is different from this line of work in the sense that its goal is to detect SCVs.

7 Conclusion

This paper proposed Dissector, a hardware software co-designed SCV detector for a typical TSO machine. Dissector hardware works by piggybacking information about pending stores with cache coherence messages. Later, it detects if any of those pending stores cause an SCV cycle. The post processing software filters out false positives and extracts detail debugging information. Dissector hardware is very lightweight, does not generate any extra network message and seamlessly handles speculatively executed loads. Our results showed that Dissector has better SCV detection ability than a state-of-the-art hardware based SCV detector. Our experiments found a previously undiscovered SCV in *fmm*. Dissector induces a negligible execution overhead of 0.02% which remains the same for more processors. Finally, it requires 3.5KB/core extra hardware.

References

1. Intel Cilk Plus. `http://cilkplus.org/`
2. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. Western Reseach Laboratory-Compaq. Research Report 95/7 (September 1995)
3. Blundell, C., Martin, M.M., Wenisch, T.F.: Invisifence: performance-transparent memory ordering in conventional multiprocessors. In: ISCA (2009)

4. Bonomi, F., et al.: An improved construction for counting Bloom filters. In: Ann. Euro. Symp. on Algo. (Sep 2006)
5. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
6. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: TACAS (2011)
7. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA (2011)
8. Duan, Y., Feng, X., Wang, L., Zhang, C., Yew, P.C.: Detecting and eliminating potential violations of sequential consistency for concurrent c/c++ programs. In: CGO (2009)
9. Gharachorloo, K., Gibbons, P.B.: Detecting violations of sequential consistency. In: SPAA (1991)
10. Gharachorloo, K., Gupta, A., Hennessy, J.: Two techniques to enhance the performance of memory consistency models. In: ICPP (1991)
11. Gniady, C., Falsafi, B., Vijaykumar, T.N.: Is sc + ilp = rc? In: ISCA (1999)
12. Gopalakrishnan, G., Yang, Y., Sivaraj, H.: Qb or not qb: An efficient execution verification tool for memory orderings. In: CAV (2004)
13. Intel: Intel parallel studio. `https://software.intel.com/en-us/intel-parallel-studio-xe`
14. Islam, M., Muzahid, A.: Characterizing real world bugs causing sequential consistency violations. In: HotPar (June 2013)
15. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computer 28(9) (Sep 1979)
16. Lin, C., Nagarajan, V., Gupta, R., Rajaram, B.: Efficient sequential consistency via conflict ordering. In: ASPLOS (2012)
17. Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm, H.J.: Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In: ISCA (2010)
18. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI (2005)
19. Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: DRFx: a simple and efficient memory model for concurrent programming languages. In: PLDI (2010)
20. Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S.: A case for an sc-preserving compiler. In: Proceedings of the 32th ACM SIGPLAN conference on Programming language and implementation. PLDI '11 (2011)
21. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI (2008)
22. Muzahid, A., Qi, S., Torrellas, J.: Vulcan: Hardware support for detecting sequential consistency violations dynamically. In: MICRO (December 2012)
23. Muzahid, A., Suárez, D., Qi, S., Torrellas, J.: Sigrace: signature-based data race detection. In: ISCA (2009)
24. Qian, X., Sahelices, B., Torrellas, J., Qian, D.: Volition: Precise and Scalable Sequential Consistency Violation Detection. In: ASPLOS (March 2013)
25. Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC Simulator (January 2005), http://sesc.sourceforge.net
26. Sen, K.: Race directed random testing of concurrent programs. In: PLDI (2008)
27. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7) (Jul 2010)
28. Shasha, D., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Transactions on Programming Languages and Systems 10(2) (Apr 1988)
29. Singh, A., Narayanasamy, S., Marino, D., Millstein, T., Musuvathi, M.: End-to-end sequential consistency. In: ISCA (2012)
30. Weaver, D., Germond, T.: The SPARC Architecture Manual Version 9. Prentice Hall, Englewood Cliffs, N.J. (1994)
31. Wenisch, T.F., Ailamaki, A., Falsafi, B., Moshovos, A.: Mechanisms for store-wait-free multiprocessors. In: ISCA (2007)
32. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: A framework for axiomatic and executable specifications of memory consistency models. In: IPDPS (2003)