Efficient Sequential Consistency via Conflict Ordering

Changhui Lin
University of California Riverside
linc@cs.ucr.edu

Vijay Nagarajan
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Rajiv Gupta
University of California Riverside
gupta@cs.ucr.edu

Bharghava Rajaram
University of Edinburgh
r.bharghava@ed.ac.uk

Abstract

Although the sequential consistency (SC) model is the most intuitive, processor designers often choose to support relaxed memory consistency models for higher performance. This is because SC implementations that match the performance of relaxed memory models require post-retirement speculation and its associated hardware costs. In this paper we propose an efficient approach for enforcing SC without requiring post-retirement speculation. While prior SC implementations guarantee SC by explicitly completing memory operations within a processor in program order, we guarantee SC by completing conflicting memory operations, within and across processors, in an order that is consistent with the program order. More specifically, we identify those conflicting memory operations whose ordering is critical for the maintenance of SC and explicitly order them. This allows us to safely (non-speculatively) complete memory operations past pending writes, thus reducing memory ordering stalls. Our experiments with SPLASH-2 programs show that SC can be achieved efficiently, with performance comparable to RMO (relaxed memory order).

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

General Terms Design, Performance, Experimentation

Keywords Sequential consistency, Conflict ordering

1. Introduction

While parallel architectures are becoming ubiquitous, extracting performance from them is contingent on programmers writing parallel software. To this end, there has been significant research on developing programming models, memory models, and debugging tools for making programmers' tasks easier. In particular, one reason why programmers find parallel programming hard is because of the intricacies involved in the underlying memory consistency models. The complexity of the memory models is well illustrated in recent work by Sewell *et al.* [28] in which they describe how the recent Intel and AMD memory model specifications do not match with the actual behavior observed in real machines. Among the various memory consistency models, the *sequential consistency* (SC) model in which memory operations appear to take place in the order specified by the program is most intuitive to programmers. Indeed, most works on semantics and software checking that strive to make concurrent programming easier assume SC [28]; several debugging tools for parallel programs, e.g. *RaceFuzzer* [27], also assume SC. Finally, SC would simplify the memory models of languages such as Java and C++ by enabling simpler implementations of Java `volatile` or C++ `atomic`.

(Prior SC implementations). In spite of the advantages of the SC model, processor designers typically choose to support relaxed consistency models; none of the Intel, AMD, ARM, Itanium, SPARC, or PowerPC processor families choose to support SC. This is because SC requires reads and writes to be ordered in program order, which can cause significant performance overhead. Indeed, SC requires the enforcement of all four possible program orderings: $r \rightarrow r$, $r \rightarrow w$, $w \rightarrow w$, $w \rightarrow r$, where r denotes a read operation and w denotes a write operation. A straightforward SC implementation enforces these orderings by delaying the next memory operation until the previous one completes, introducing a significant number of program ordering stalls. Some of these stalls can be reduced in a dynamically scheduled ILP processor, where *in-window speculation* support can be used to execute memory operations out-of-order while completing them in program order [12]. Nonetheless, enforcing the $w \rightarrow r$ (and $w \rightarrow w$) order necessitates that the write-buffer is drained before a subsequent memory operation can be completed. Thus, a high latency write can still cause significant program ordering stalls – which in-window speculation is unable to hide. Recent works have utilized *post-retirement speculation* – speculation beyond instruction window – to eliminate these program ordering stalls [6–8, 11, 13–15, 23, 31]. While this approach is promising, the significant hardware complexity associated with post-retirement speculation could hinder its wide-spread adoption [3].

(Our Approach). SC requires memory operations of all processors to appear to perform in some *global memory order*, such that, memory operations of each processor appear in this global memory order in the order specified by the program [17]. Prior SC implementations ensure this by enforcing program ordering by explicitly completing memory operations in program order. More specifically, if m_1 and m_2 are two memory operations with m_1 pre-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03. . . $10.00

ceding m_2 in the program order, prior SC implementations ensure $m_1 \rightarrow m_2$ by allowing m_2 to complete only after m_1 completes; in effect, m_2 is made to wait until m_1 and all of m_1's predecessors in the global memory order have completed. In other words, if $pred(m_1)$ refers to m_1 and its predecessors in the global memory order, m_2 is allowed to complete only when all memory operations from $pred(m_1)$ have completed.

While enforcing program ordering is a sufficient condition for ensuring SC [26], it is not a necessary condition [19, 25, 29]. In contrast with prior SC implementations, we ensure SC by *explicitly ordering only conflicting memory operations* – where conflicting memory operations are pairs of memory operations that access the same address, with one of them being a write. More specifically, we observe that m_1 *appears* to be ordered before m_2, as long as those memory operations from $pred(m_1)$ which conflict with m_2, complete before m_2. In other words, if $pred_p(m_1)$ refers to memory operations from $pred(m_1)$ which are pending completion, m_2 can safely complete before m_1 as long as m_2 does not conflict with any of the memory operations from $pred_p(m_1)$. It is worth noting that the above condition generally evaluates to true, since in well-written parallel programs conflicting accesses are relatively rare and staggered in time [14, 19].

Our SC implementation, which allows for memory operations to be safely (and non-speculatively) completed past pending writes, is considered in the context of a scalable CMP with local caches kept coherent using an invalidation based cache coherence protocol. When a write-miss w is encountered, we conservatively estimate $pred_p(w)$ to be the set of write misses that are currently being serviced by the coherence controller. This lets us safely complete subsequent memory operations which do not conflict with $pred_p(w)$, even while w is pending.

The main contribution of this paper is an efficient and lightweight approach for SC using conflict ordering. More specifically, our SC implementation is:

- **Efficient.** Our experiments with SPLASH-2 programs show that SC can be achieved efficiently, with performance comparable to RMO (relaxed memory order).
- **Lightweight.** Ours is the first SC implementation that performs almost as well as RMO, while requiring no post-retirement speculation.
- **Scalable.** Our SC implementation scales well, with performance that is comparable to RMO even with 32 cores.

2. Background

Memory consistency models [4] constrain memory behavior with respect to read and write operations from multiple processors, providing formal specifications of how the memory system will appear to the programmer. Each of these models offers a trade-off between programming simplicity and high performance. Sequential consistency (SC) is the simplest model that is easy to understand but can potentially affect the performance of programs. The strict memory ordering imposed by SC can potentially restrict both compiler and hardware optimizations that are possible in uniprocessors [4]. For performance reasons, researchers have proposed relaxed memory consistency models at the cost of programming complexity [4, 5]. Such memory models provide fence instructions to enable programmers to explicitly provide memory ordering. For instance, the SPARC RMO model (relaxed memory order) relaxes all four program orderings – but provides memory fence instructions that the programmer can use for overriding the relaxations and enforce memory orderings on demand. The complexities of specifying relaxed memory models, however, combined with the additional burden it places on the programmers have rendered relaxed memory models inadequate [3]. Next, we discuss different approaches for achieving SC.

2.1 SC via program ordering

In program ordering based SC implementations the hardware directly ensures all four possible program ordering constraints [6, 12–14, 23]; they are based on the seminal work by Scheurich and Dubois [26], in which they state the sufficient conditions for SC for general systems with caches and interconnection networks.

(Naive). A naive way to enforce program ordering between a pair of memory operations is to delay the second until the first fully completes. However, this can result in a significant number of memory ordering stalls; for instance, if the first access is a miss, then the second access has to wait until the miss is serviced.

(In-window speculation). Out-of-order processing capabilities of a modern processor can be leveraged to reduce some of these stalls. This is based on the observation that memory operations can be freely reordered as long as the reordering is not observed by other processors. Instead of waiting for an earlier memory operation to complete, the processor can use hardware prefetching and speculation to execute memory operations out of order, while still completing in order [12, 33]. However, such reordering is within the instruction window – where the instruction window refers to the set of instructions that are in-flight. If the processor receives an external coherence (or replacement) request for a memory operation that has executed out of order, the processor's recovery mechanism is triggered to redo computation starting from that memory operation. Nonetheless, enforcing the $w \rightarrow r$ (and $w \rightarrow w$) order necessitates that the write-buffer is drained before a subsequent memory operation can be completed. Thus, a high latency write can still cause significant program ordering stalls – which in-window speculation is unable to hide. Our experiments with the SPLASH-2 programs show that programs spend more than 20% of their execution time on average waiting for the write buffer to be drained. For our work, we assume in-window speculation support as part of baseline implementations of SC as well as fences in RMO.

(Post-retirement speculation). Since in-window speculation is not sufficiently able to reduce program ordering stalls, researchers have proposed more aggressive speculative memory reordering techniques [6–8, 11, 13–15, 23, 31]. The key idea is to *speculatively retire* instructions neglecting program ordering constraints while maintaining the state of speculatively-retired instructions separately. One way to do this is to maintain the state of speculatively-retired instructions at a fine granularity, which enables precise recovery from misspeculations [13, 14, 23]. This obviates the need for a load to wait until the write buffer is drained and is made to retire speculatively. More recently, researchers have proposed *chunk based* techniques [6–8, 11, 15, 31] which again use aggressive speculation to efficiently enforce SC at the granularity of coarse-grained chunks of instructions instead of individual instructions. While the above approaches show much promise, hardware complexity associated with aggressive speculation, being contrary to the design philosophy of multi-cores consisting of simple energy-efficient cores [1, 2], can hinder wide-spread adoption. Through our work, we seek to show that a lightweight SC implementation is possible without sacrificing performance.

2.2 Other SC approaches

Shasha and Snir, in their seminal work [29], observed that not all pairs of memory operations need to be ordered for SC; only those pairs which conflict with others run the risk of SC violation and consequently need to be ordered. To ensure memory operation pairs are not reordered, memory fences are inserted. They then propose *delay set analysis*, a compiler based algorithm for minimizing the

number of fences inserted for ensuring SC. This work led to other compiler based algorithms that implement various fence insertion and optimization algorithms [10, 18]. Recently, we proposed *conditional fence* [19], a novel fence mechanism that dynamically decides if a fence needs to stall; we show that with conditional fences, the number of fences that actually need to stall is reduced significantly. The limitation of compiler-based SC implementations is that they rely on alias analysis for identifying conflicts, and so are overly conservative for programs that employ pointers and dynamic data structures [3, 19].

In developing delay set analysis, Shasha and Snir introduced a precise formalism for SC in terms of the program order being consistent with the execution order. Their SC formalism reflects the notion that SC does not actually require memory operations to be performed in program order; it merely requires the execution to be consistent with program order. Gharachorloo *et al.*later built on Shasha and Snir's formalism to derive aggressive SC specifications [25] which relaxes program ordering constraints. While our idea of conflict ordering and its formalization is strongly influenced by Shasha and Snir's SC formalism, we additionally leverage this idea and propose a lightweight and efficient SC implementation, that performs almost as well as RMO.

Mainstream programming languages like C++ and Java use variants of the data-race-free memory model [4, 5] which guarantee SC as long as the program is free of data races. However, if the programs do have data races then these models provide much weaker semantics. To redress this there have been works that employ dynamic race detection [9] in order to stop execution when semantics become undefined. Since dynamic data race detection can be slow, recent works [20, 21, 30] propose raising an exception upon encountering an SC violation as this can be done more efficiently.

Finally, recent work has shown that it is possible to support high performance compiler that preserves SC [22]. Their SC-preserving compiler, however, cannot prevent the hardware from exposing non-SC behaviour. Thus, our work which proposes an efficient and lightweight hardware SC implementation, is complementary to the above work.

3. SC via conflict ordering

In this section, we first informally describe our approach of enforcing SC using conflict ordering with a motivating example. We then formally prove that our approach correctly enforces SC.

3.1 A motivating example

SC requires that memory operations appear to complete in program order. To ensure this, prior SC implementations force memory operations to *explicitly* complete in program order. However, is program ordering necessary for ensuring SC? Let us consider the example in Fig. 1, which shows memory operations a_1, a_2, b_1 and b_2 from processors A and B. They are two pairs of conflicting accesses. Can a_2 complete before a_1 in an execution without breaking SC? Prior SC implementations forbid this and force a_2 to wait until a_1 completes. In this work, we observe that a_1 *appears* to complete before a_2, as long as a_2 is made to complete after b_1, with which a_2 conflicts. In other words, a_2 can complete before a_1 without breaking SC, as long as we ensure that a_2 waits for b_1 to complete. Indeed, if b_1 and b_2 have completed before a_1 and a_2, the execution order (b_1, b_2, a_2, a_1) is equivalent to the original SC order (b_1, b_2, a_1, a_2) as the values read in the two executions are identical.

We propose *conflict ordering*, a novel approach to SC, in which SC is enforced by explicitly ordering only conflicting memory operations. Conflict ordering allows a memory operation to complete, as long as all conflicting memory operations prior to it in the global memory order have completed. More specifically, let m_1 and m_2 be consecutive memory operations; furthermore let $pred(m_1)$ refer

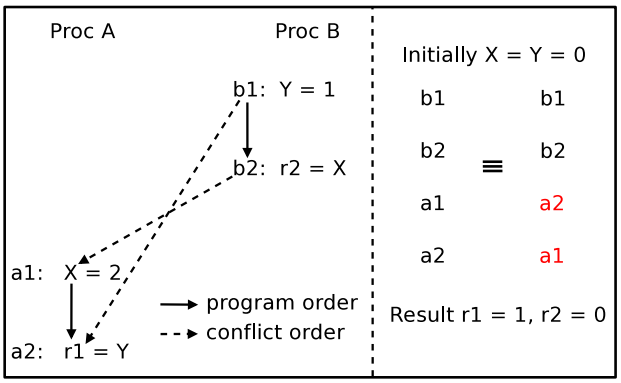

Figure 1. A motivating example: As long as a_2 is made to wait until the conflicting access b_1 completes, a_2 can be safely reordered before a_1, without breaking SC

to memory operations that include m_1 and those before m_1 in the global memory order. Conflict ordering allows m_2 to safely complete as long as those memory operations from $pred(m_1)$, which conflict with m_2, have completed.

3.2 Conflict ordering enforces SC

In this section, we prove that conflict ordering enforces SC using the formalism of Shasha and Snir [29]. For the following discussion, we assume an invalidation based cache coherence protocol for processors with private caches. A read operation is said to complete when the returned value is bound and can not be updated by other writes; a write operation is said to complete when the write invalidates all cached copies, and the generated invalidates are acknowledged [12]. Furthermore, we assume that the coherence protocol serializes writes to the same location and also ensures that the value of a write not be returned by a read until the write completes – in other words, we assume that the coherence protocol ensures *write atomicity* [4].

Definition 1. *The program order* **P** *is a local (per-processor) total order which specifies the order in which the memory operations appear in the program. That is,* $m_1 \mathbf{P} m_2$ *iff the memory operation* m_1 *occurs before* m_2 *in the program.*

Definition 2. *The conflict relation* **C** *is a symmetric relation on* **M** *(all memory operations) that relates two memory operations (of which one is a write) which access the same address.*

Definition 3. *An execution* **E** *(or execution order or conflict order) is an orientation of* **C***. If* $m_1 \mathbf{C} m_2$*, then either* $m_1 \mathbf{E} m_2$ *or* $m_2 \mathbf{E} m_1$ *holds.*

Definition 4. *An execution* **E** *is said to be atomic, iff* **E** *is a proper orientation of* **C***, that is, iff* **E** *is acyclic.*

Definition 5. *An execution* **E** *is said to be sequentially consistent, iff* **E** *is consistent with the program order* **P***, that is, iff* $\mathbf{P} \cup \mathbf{E}$ *has no cycles.*

Definition 6. *The global memory order* **G** *is the transitive closure of the program order and the execution order,* $\mathbf{G} = (\mathbf{P} \cup \mathbf{E})^+$.

Remark. In the scenario shown in Fig. 1, b_1 appears before a_1 in the global memory order, since b_1 appears before b_2 in program order, and b_2 is ordered before a_1 in the execution order. That is, $b_1 \mathbf{G} a_1$ since $b_1 \mathbf{P} b_2$ and $b_2 \mathbf{E} a_1$.

Definition 7. *The function* $pred(m)$ *returns a set of memory operations that appear before* m *in the global memory order* **G***, including* m*. That is,* $pred(m) = \{m\} \cup \{m' : m' \mathbf{G} m\}$.

Definition 8. *An SC implementation constrains the execution by enforcing certain conditions under which a memory operation can be completed. An SC implementation is said to be correct if it is guaranteed to generate an execution that is sequentially consistent.*

Definition 9. *Conflict ordering is our proposed SC implementation in which a memory operation m_2 (whose immediate predecessor in program order is m_1) is allowed to complete iff those memory operations from $pred(m_1)$ which conflict with m_2 have completed. That is, m_2 is allowed to complete iff $\{m \in pred(m_1) : m\mathbf{C}m_2\}$ have already completed.*

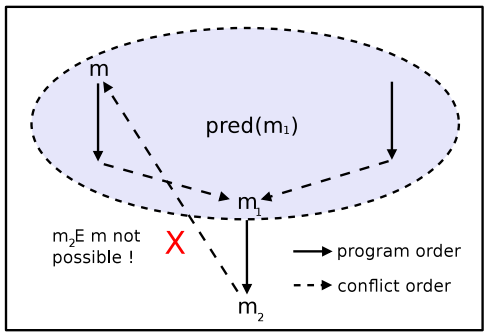

Figure 2. Conflict ordering ensures SC

Lemma 1. *Any execution* **E** *(Definition 3) is atomic.*

Proof. Let w and r with a subscript be a write operation and a read operation. Write-serialization ensures that $w_1\mathbf{E}w_2$ iff w_1 completes before w_2; furthermore, $w_1\mathbf{E}w_2$ and $w_2\mathbf{E}w_3$ result in $w_1\mathbf{E}w_3$. Write-atomicity also ensures that $w_1\mathbf{E}r_2$ iff w_1 completes before r_2. Since reads are atomic, $r_1\mathbf{E}w_2$ iff r_1 completes before w_2. Furthermore, $w_1\mathbf{E}r_2$ and $r_2\mathbf{E}w_3$ result in $w_1\mathbf{E}w_3$. Thus, **E** is acyclic; therefore, from Definition 4, **E** is atomic. □

Theorem 1. *Conflict ordering is correct.*

Proof. We need to prove that any execution **E** generated by conflict ordering is sequentially consistent. That is, to prove that the graph $\mathbf{P} \cup \mathbf{E}$ is acyclic (from Definition 5). Let us attempt a proof by contradiction, and assume that there is a cycle in $\mathbf{P} \cup \mathbf{E}$. Since **E** is acyclic (from Lemma 1), the assumed cycle should contain at least one program order edge. Without loss of generality, let us assume that the program order edge that contains a cycle is $m_1\mathbf{P}m_2$ as shown in Fig. 2. To complete the cycle, there must be a conflict order edge $m_2\mathbf{E}m$, where $m \in pred(m_1)$. Conflict ordering, however, ensures that those memory operations from $pred(m_1)$, which conflict with m_2 would have completed before m_2 (from Definition 9). Since $m \in pred(m_1)$, m would have completed before m_2. Thus, the conflict order edge $m_2\mathbf{E}m$ is not possible, which results in a contradiction. Thus, conflict ordering is correct. □

4. Hardware Design

In this section we describe our hardware design that incorporates conflict ordering to enforce SC efficiently. We first describe our basic design in which all memory operations will have to check for conflicts before they can complete. We then describe how we can determine phases in program execution where memory operations can complete without checking for conflicts.

(System Model). For the following discussion we assume a tiled chip multiprocessor, with each tile consisting of a processor, a local L1 cache and a single bank of the shared L2 cache. We assume that the local L1 caches are kept coherent using a directory based cache coherence protocol, with the directory distributed over the L2 cache. We assume that addresses are distributed across the directory at a page granularity using the first touch policy. We assume a directory protocol in which the requester notifies the directory on transaction completion, so each coherence transaction can have a maximum of four steps. Thus, each coherence transaction remains active in the directory until the time at which it receives the notification of transaction completion. Furthermore, we assume that the cache coherence protocol provides write atomicity. We assume each processor core to be a dynamically scheduled ILP processor with a reorder buffer (ROB) which supports in-window speculation. All instructions, except memory writes, are made to complete in program order, as and when they retire from the ROB. Writes on the other hand, are made to retire into the write-buffer, so that the processor need not wait for them to complete. Finally, it is worth noting that our proposal is also applicable to bus based coherence protocols; indeed, we report experimental results for both bus based and directory based protocols.

4.1 Basic conflict ordering

We describe our conflict ordering implementation which allows memory operations (both reads and writes) to complete past pending writes, while ensuring SC; it is worth noting, however, that our system model does not allow memory operations to complete past pending reads. Recall that conflict ordering allows a memory operation m_2, whose immediate predecessor is m_1, to complete as long as m_2 does not conflict with those memory operations from $pred(m_1)$ which are pending completion – let us call such pending operations as $pred_p(m_1)$. Thus, for m_2 to safely complete, we need to be sure that m_2 does not conflict with any of the memory operations from $pred_p(m_1)$. Alternatively, if $addr_p(m_1)$ refers to the set of addresses accessed by memory operations from $pred_p(m_1)$, we can safely allow m_2 to complete if its address is not contained in $addr_p(m_1)$. The challenge is to determine $addr_p(m_1)$ as quickly as possibly – and in particular without waiting for m_1 to complete. Next, we show how we compute $addr_p(m_1)$ for a write-miss, cache-hit, and a read-miss respectively.

(Write-misses). Our key idea for determining $addr_p(m_1)$ for a write-miss m_1 is to simply get this information from the directory. We show that those memory operations from $pred_p(m_1)$, if any, would be present in the directory (as pending memory operations that are currently being serviced in the directory), provided write-misses such as m_1 are issued to the directory, before subsequent memory operations are allowed to complete. In other words, if *addr-list* refers to the set of addresses of the cache misses being serviced in the directory, $addr_p(m_1)$ would surely be contained in *addr-list*. In addition to this, since our system model does not allow memory operations to complete past pending reads, we are able to show that $addr_p(m_1)$ would surely be contained in *write-list* – where *write-list* refers to the set of memory addresses of the write-misses being serviced in the directory. Consequently, when the write-miss m_1 is issued to the directory, the directory replies back with the *write-list*, containing the addresses of the write-misses which are currently being serviced by the directory (to minimize network traffic we safely approximate the write-list by using a bloom filter). A subsequent memory operation m_2 is allowed to complete, only if m_2 does not conflict with any of the writes from the write-list. If m_2 does conflict, then the local cache block m_2 maps to is invalidated, and the memory operation m_2 and its following instructions are replayed when necessary. During replay, m_2 would turn out to be a cache miss and hence the miss request would be sent to the directory. This would ensure that m_2 would be ordered after its conflicting write that was pending in the

directory. *It is worth noting that memory operations that follow a pending write, will now need to wait only until the write is issued to the directory and get a reply back from the directory, as opposed to waiting for the write-miss to complete.* While the former takes as much time as the round trip latency to the directory, the latter can take a significantly longer time – since it may involve the time taken to invalidate all shared copies, and also access memory if it is a miss.

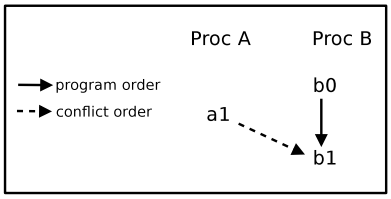

Figure 3. $pred_p(b_1) = \{b_1\} \cup pred_p(b_0) \cup pred_p(a_1)$.

(Cache-hits). Fig. 3 shows how $pred_p(b_1)$ relates to $pred_p(b_0)$, where b_0 is the immediate predecessor of b_1 in the program order, and a_1 is the immediate predecessor of b_1 in the conflict order. As we can see, $pred_p(b_1) = \{b_1\} \cup pred_p(b_0) \cup pred_p(a_1)$; this is because the global memory order is the union of the program order and the conflict order. If, however, b_1 is a cache hit, then its immediate predecessor in the conflict order must have completed and hence cannot be pending; if it were pending, b_1 would become a cache-miss. Thus $pred_p(b_1) = \{b_1\} \cup pred_p(b_0)$. Since memory operations that follow a cache-hit are allowed to complete only after the cache hit completes, $pred_p(b_1)$ remains the same as $pred_p(b_0)$. Consequently, $addr_p(b_1)$ remains the same as $addr_p(b_0)$ and thus, the write-list for a cache-hit need not be computed afresh.

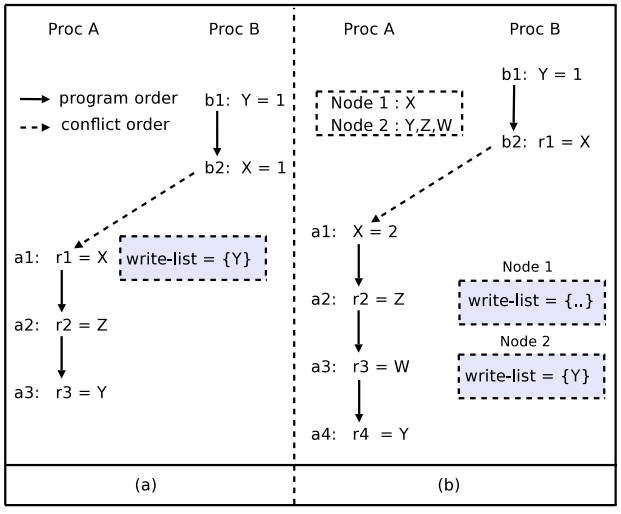

Figure 4. Basic Conflict Ordering: Example

(Read-misses). One important consequence of completing memory operations past pending writes is that, when a read-miss completes, there might be writes before it in the global memory order that are still pending. As shown in Fig. 4(a), if b_2 is allowed to complete before b_1, b_1 might still be pending when a_1 completes. Now, conflict ordering mandates that memory operations that follow a_1 can complete, only when they do not conflict with $pred_p(a_1)$. To enable this check, read-misses are also made to consult the directory and determine the write-list. Accordingly, when read-miss a_1 consults the directory, it fetches the write-list and replies to the processor. Having obtained the write-list, a_2 which does not conflict with memory operations from the write-list, is allowed to complete. Memory operation a_3, since it conflicts, is replayed obtaining its value from b_1 via the directory.

4.1.1 Handling distributed directories

To avoid a single point of contention, directories are typically distributed across the tiles, with the *directory placement policy* deciding the mapping from the address to the corresponding *home* directory; we assume the *first touch* directory placement policy. With a distributed directory, a write-miss (or a read-miss) m_1 is issued to its home directory and can only obtain the list of pending writes from *that* directory. This means that a memory operation m_2 that follows m_1 can use the write-list to check for conflicts only if m_2 maps to the same home directory as m_1. If m_2 does not map to the same home node as m_1, then m_2 will have to be conservatively assumed to be conflicting. Consequently m_2 will have to go to its own home directory to ensure that it does not conflict. If it does not conflict, then m_2 can simply commit like a cache hit; otherwise, it is treated like a miss and replayed. When m_2 goes to its home directory to check for conflicts, we take this opportunity to fetch the write-list from m_2's home directory, so that future accesses to the same node can check for conflicts locally. To accommodate this, the local processor has multiple write-list registers which are tagged by the tile id.

The scenario shown in Fig. 4(b) illustrates this. Let us assume that the variable X maps to Node 1, while variables Z, W and Y map to Node 2. Note that the processor also has two write-list registers. When a_1 is issued to its directory, it returns the pending write-misses from Node 1. Consequently, this is put into one of the write-list register which is tagged with the id of Node 1. The contents of the other write-registers are invalidated as they might contain out-of-date data. When a_2 tries to commit, the tags of the write-list registers are checked to see if any of the write-list registers is tagged with the id of Node 2, which is the home node for a_2. Since none of the write-list registers have the contents of Node 2, a_2 is sent to its home directory (Node 2) to check for conflicts. After ensuring that a_2 does not conflict, the pending stores of the home directory (from Node 2) are returned and inserted into the write-list register. This allows us to commit a_3 which maps to the same node. Likewise, when a_4 tries to commit, it is found to be conflicting and hence replayed.

Although the distributed directory could potentially reduce the number of memory operations that can complete past pending writes, our experiments show that, due to locality, most of the nearby accesses tend to map to the same home node, which allows us to complete most memory operations past pending writes.

4.1.2 Correctness

The correctness of our conflict ordering implementation hinges on the fact that when a cache-miss m_1 is issued to the directory, the write-list returned by the directory is correct; that is, the write-list should include $addr_p(m_1)$, the addresses of all pending memory operations that occur before m_1 in the global memory order. We prove this formally, next.

Lemma 2. *When a cache-miss m_1 is issued to the directory, all pending memory operations prior to it in the global memory order must be present in the directory . That is, when m_1 is issued to the directory, $\{m : m\mathbf{G}m_1\}$ must be present in the directory.*

Proof. (1) m_1 is a write-miss. When m_1 is issued to the directory, conflict ordering ensures that all pending memory operations prior to m_1 in the program order must have been issued to the directory. (2) m_1 is a read-miss. m_1 is issued to the directory to fetch the

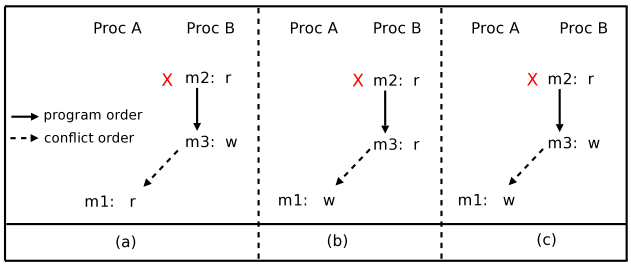

Figure 5. Correctness of conflict ordering implementation: m_2 cannot be a read in each of the 3 scenarios

write-list only when it is retired, and all memory operations prior to it have in turn been issued. Thus, whether m_1 is a write-miss or a read-miss, when m_1 is issued to the directory, $\{m : m\mathbf{P}m_1\}$ must be present in the directory. Likewise, when m_1 is issued to the directory all memory operations prior to m_1 in the conflict order, $\{m : m\mathbf{E}m_1\}$ must have been issued to the directory. Thus, when m_1 is issued to the directory, $\{m : m\mathbf{G}m_1\}$ must be present in the directory. □

Lemma 3. *When a read-miss m_1 is issued to the directory (to obtain the write-list), none of the pending memory operations prior to it in the global memory order are read-misses.*

Proof. Let us attempt a proof by contradiction and assume that such a pending read-miss exists. Let m_2 be the assumed read-miss such that $m_2 \in pred_p(m_1)$. Now the read-miss m_2 cannot occur before m_1 in the program order, as all such read-misses would have retired and hence cannot be pending. m_2 cannot occur before m_1 in the conflict order, as two reads do not conflict with each other. Thus, the only possibility is as shown in Fig. 5(a), where there is a write-miss m_3 such that $m_2\mathbf{P}m_3$ and $m_3\mathbf{E}m_1$ (without loss of generality). This again, however, is impossible since the write m_3 will be issued only after all reads before it (including m_2) complete. □

Lemma 4. *When a write-miss m_1 is issued to the directory, all pending read-misses prior to it in the global memory order, conflict with m_1.*

Proof. As shown in Fig. 5(b), it is possible that there exists m_3, a read-miss, such that $m_3\mathbf{G}m_1$ since $m_3\mathbf{E}m_1$. There cannot be, however, any read-miss m_2 such that $m_2\mathbf{G}m_1$ with m_2 not conflicting with m_1. Proof is by contradiction, along similar lines to Lemma 3 (m_2 in Fig. 5 (b) and Fig. 5 (c) cannot be reads). □

Theorem 2. *When a cache-miss m_1 is issued to the directory, the write-list returned will contain $addr_p(m_1)$, the addresses of all pending memory operations that occur before m_1 in the global memory order.*

Proof. When a cache-miss m_1 is issued to the directory, the addresses of the memory operations serviced in the directory (addr-list) will contain $addr_p(m_1)$ (from lemma 2). All pending read-misses which occur before m_1 in the global memory order, if any, would conflict with m_1 and hence access the same address as m_1. (from Lemma 3 and Lemma 4). Thus, the write-list is guaranteed to contain $addr_p(m_1)$. □

4.2 Enhanced conflict ordering

In this section, we describe our enhanced conflict ordering design, in which we identify phases in program execution where memory operations can complete without checking for conflicts. But first, we discuss the limitations of basic conflict ordering, to motivate our approach.

4.2.1 Limitations of basic conflict ordering

We illustrate the limitations of basic conflict ordering with the example shown in Fig. 6(a) and (b). As we can see, in Fig. 6(a), the write-miss a_1 estimates $pred_p(a_1)$ by consulting the directory and obtaining the write-list. All memory operations that follow the pending write miss a_1 need to be checked with the write-list for a conflict, before they can complete. It is important to note that even after a_1 completes, memory operations following a_1 will still have to be checked for conflicts. This is because b_1, which precedes a_1 in the global memory order, could still be pending when a_1 completes. In other words, the completion of the store a_1 is no longer an indicator of all memory operations belonging to $pred(a_1)$ completing. Thus, in basic conflict ordering, all memory operations that follow a write miss (or a read miss) need to be continually checked for conflicts before they can safely complete.

Another limitation stems from the fact that a read-miss (or a write-miss) *conservatively* estimates its predecessors in the global memory order by accessing the directory. Let us consider the scenario shown in Fig. 6(b), in which the read-miss a_1 estimates $pred_p(a_1)$ from the directory. However, if b_2 and a_1 are sufficiently staggered in time, which is typically common [19], then all memory operations belonging to $pred(b_2)$ (including b_1) might have already completed by the time a_1 is issued. In such a case, there is no need for a_1 to compute its write list; indeed, memory operations such as a_2 can be safely completed past a_1.

4.2.2 Our approach

(HW support and operation). Our approach is to keep track of the (local) memory operations that have retired from the ROB, but whose predecessors in the global memory order are still pending. We call such memory operations as *active* and we track such memory operations in a per-processor *augmented write buffer* (AWB). The AWB is like a normal write-buffer, in that, it buffers write-misses; unlike a conventional write-buffer, however, it also buffers the addresses of other memory operations (including read-misses, read-hits and write-hits) that are active. Therefore, an empty AWB indicates an executing phase in which all preceding memory operations in the global memory order have completed; this allows us to complete succeeding memory operations without checking for conflicts. To reduce the space used by AWB, the consecutive cache hits to the same block are merged.

We now explain how we keep track of active memory operations in the AWB. A write-miss that retires from the ROB is marked active by inserting it into the AWB, as usual. The write-miss, however, is not necessarily removed from the AWB (i.e. marked *inactive*) when it completes; we will shortly explain the conditions under which a write-miss is removed from the AWB.

A memory operation which retires from the ROB while the AWB is non-empty is active by definition. Consequently, a cache hit which retires while the AWB is non-empty is marked active by inserting its cache block address into the AWB; it is subsequently removed when the memory operation becomes *inactive* – i.e., when all prior entries in the AWB have been removed.

A cache miss which retires while the AWB is non-empty, like a cache hit, is marked active by inserting its cache block address into the AWB. However, unlike a cache hit, its block address is not necessarily removed when all prior entries in the AWB have been removed. Indeed, a cache miss remains active, and hence its cache block address remains buffered in the AWB, until its predecessors in the conflict order become inactive. Accordingly, even if a write-miss completes, it remains buffered in the AWB until all the cache blocks that it invalidates turn inactive. Likewise, a read-miss that completes is inserted into the AWB, if it gets its value from a cache block that is marked active. Here, a cache block is said to be active

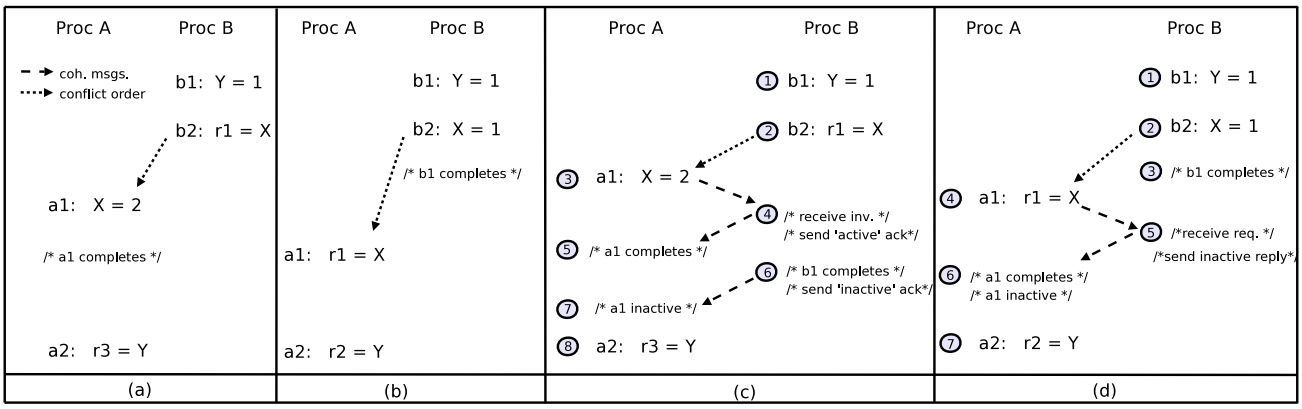

Figure 6. (a) and (b): Limitations of basic conflict ordering; (c) and (d): Our approach

if the corresponding cache block address is buffered in the AWB, and inactive otherwise.

Thus, to precisely keep track of cache misses that are active, we need to somehow infer whether its predecessors in the conflict order are active. We implement this by tagging coherence messages as active or inactive. When a processor receives a coherence request for a cache block, we check the AWB to see if it is marked active. If it is marked active, the processor responds to this coherence request with a coherence response that is tagged active. This would enable the recipient (cache miss) to infer that the coherence response is from an active memory operation. At the same time, when a processor receives a coherence request to an active cache block, it is buffered; when the cache block eventually becomes inactive, we again respond to the buffered coherence request with a coherence response that is tagged *inactive*. This would enable the recipient (cache miss) to infer that the response is from a block that has transitioned to inactive. A cache miss, when it receives a coherence response, is allowed to complete irrespective of whether it receives an active or an inactive coherence response; however, it remains active (and hence buffered in the AWB) until it gets an inactive response. Finally, by not allowing an active cache block to be replaced, we ensure that if a cache miss has predecessors in the conflict order, it will surely be exposed via the tagged coherence messages.

(Write-Miss: Example). We now explain the operation with the scenario shown in Fig. 6(c) which addresses the limitation shown in Fig. 6(a). First, the write-miss b_1 is issued from the ROB of processor B and is inserted into the AWB (step 1), after which the read-hit b_2 is made to complete past it. Since, the AWB is non-empty when b_2 completes, b_2 is marked active by inserting the cache block address X into the AWB (step 2). Then, write-miss a_1 to same address X is issued in processor A, inserted into the AWB, and issued to the directory (step 3). The directory then services this write-miss request, sending an invalidation request for cache block address X to processor B. Since cache block address X is active in processor B (cache block address X is buffered in processor B's AWB), processor B sends an active invalidation acknowledgement back to processor A (step 4); at the same time, processor B buffers the invalidation request so that it would be able to respond again when cache block address X eventually becomes inactive. When processor A receives the invalidation acknowledgement, a_1 completes (step 5), and so it sends a completion acknowledgement to the directory. It is worth noting that a_1 has completed at this point, but is still active and hence is still buffered in processor A's AWB. Let us assume that b_1 completes at step 6; furthermore let us assume that at this point all the predecessors of b_1 in the global memory order have also completed – in other words, b_1 has becomes inactive. Since b_1 has become inactive, cache block address Y is removed from the AWB. This, in turn, causes b_2 (cache block address X) to be removed from the AWB, because all those entries that preceded b_2 (including b_1) have been removed from the AWB. Since cache block address X has become inactive, processor B again responds to the buffered invalidation request by sending an inactive invalidation acknowledgement to processor A. When processor A receives this inactive acknowledgement, the recipient a_1 is made inactive and hence removed from the AWB (step 7). This will allow succeeding memory operations like a_2 to safely complete without checking for conflicts (step 8).

(Read-Miss: Example). Fig. 6(d) addresses the limitation shown in Fig. 6(b). First, the write-miss b_1 is issued and made active by inserting it into the AWB (step 1). Then, the write-hit b_2 which completes past it is made active by inserting address X into the AWB (step 2). Let us assume that b_1 then completes, and also becomes inactive (step 3). This causes b_1 to be removed from the AWB, which in turn causes b_2 (cache block address X) to be removed. When the read-miss a_1 is issued to the directory (step 4), it sends a data value request for address X to processor B. Since the cached block address X is inactive, processor B responds with an inactive data value reply to processor A (step 5). Upon receiving this value reply, a_1 completes. Furthermore, since the reply is inactive, it indicates that all the predecessors of a_1 in the conflict order have completed. Thus, the write-list is not computed and a_1 is not inserted into the AWB. Indeed when a_2 is issued (step 8), assuming the AWB is empty, it can safely be completed.

(Misses to uncached blocks). When a cache-miss is issued to the directory and it is found to be uncached in each of the other processors, this indicates that the particular cache block is inactive in each of the other processors. This is because an active block is not allowed to be replaced from the local cache. This allows us to handle a miss to an uncached block similar to a cache hit, in that, we do not need to compute a new write-list for such misses. Indeed, if there are no other pending entries in the AWB, then memory operations that come after a miss to an uncached block can be completed without checking for conflicts. It is worth noting that misses to local variables, which account for a significant percentage of misses, would be uncached in other processors. This optimization would allow us to freely complete memory operations past such local misses.

(Avoiding AWB snoops). In the above design, all coherence requests and replacement requests must snoop the AWB first to see if the corresponding cache block is marked active. To avoid this,

we associate an *active bit* with every cache block; the active bit is set whenever the corresponding cache block address is inserted into the AWB and reset, whenever the cache block address is not contained in the AWB.

(Deadlock-freedom). Deadlock-freedom is guaranteed because memory operations that have been marked active will eventually become inactive. It is worth recalling that a memory operation becomes inactive when all its predecessors in the global memory order in turn become inactive. In theory, since conflict ordering guarantees SC, there cannot be any cycles in the global memory order, which ensures deadlock-freedom. The fact that an active cache block is not allowed to be replaced, however, can potentially cause the following subtle deadlock scenario. In this scenario, an earlier write-miss is not allowed to bring its cache block into its local cache, since all the cache blocks in its set have been marked active by later memory operations which have completed. In such a scenario, the later memory operations that have completed are marked active, and are waiting for the earlier write-miss to turn inactive; the earlier write-miss, however, cannot complete (and become inactive), since it is waiting for the later memory operations to turn inactive. We avoid such a scenario by forcing a write-miss to invalidate the cache block chosen for replacement and set its active bit, before the write-miss is issued to the directory. This will ensure that later memory operations which complete before the write-miss will not be able to use the same cache block used by the write-miss, as this cache block has been marked active by the write-miss. Thus, the deadlock is prevented.

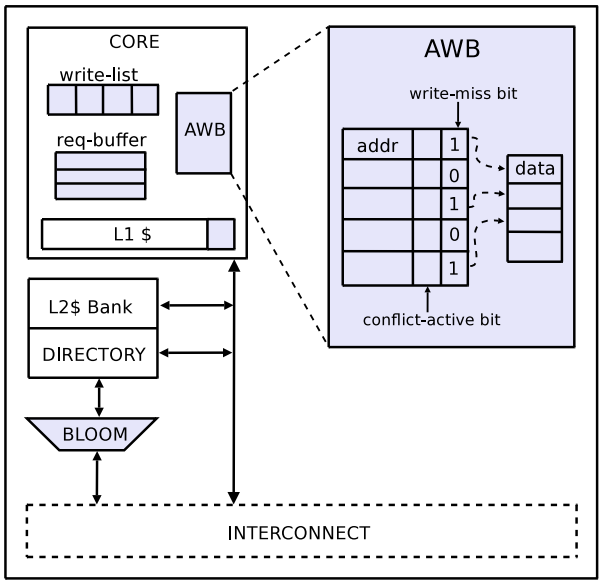

Figure 7. Hardware Support: Conflict Ordering

4.3 Summary

(Hardware support). Fig. 7 summarizes the hardware support required by conflict ordering. First, each processor core is associated with a set of write-list registers. Second, each processor core is associated with an augmented write buffer (AWB), which replaces a conventional write-buffer. Like a conventional write-buffer, each entry of the AWB is tagged by the cache block address; unlike a conventional write buffer, however, it also buffers read-misses, read-hits and write-hits. To distinguish write-misses from other memory operations, each entry of the AWB is associated with a *write-miss bit*. If the current entry corresponds to a write-miss, i.e. the write-miss bit is set, the entry additionally points to the data that is written by the current write-miss. Each entry is also associated with a *conflict-active* bit, which is set if any of its predecessors which conflict with it are active. Third, an *active* bit is added to each local cache block, which indicates whether that particular cache block is active. Lastly, conflict ordering requires minor extensions to the cache coherence subsystem. Each processor is associated with a *req-buffer* which is used to buffer coherence requests to active cache blocks. Each coherence response message (data value reply and invalidation acknowledgement) is tagged with an active bit, to indicate whether it is an active response or an inactive response. Also, conflict ordering involves the exchange of additional coherence message. While additional coherence messages are exchanged as part of conflict ordering, it is worth noting that the coherence protocol and its associated transactions for maintaining coherence are left unchanged. We summarize the additional transactions next.

(Write-miss actions). The actions performed when the write-miss retires from the ROB are as follows:

- Insert into the AWB. The write-miss is first inserted into the AWB as follows: the inserted entry is tagged with the cache block address of the write-miss; since the entry corresponds to a write-miss, the write-miss bit is set to 1 and the entry points to the data written by the write-miss; the conflict-active bit is set to 1 since its predecessor in the conflict order may be active.
- Invalidate replacement victim. Once the write-miss is inserted into the AWB, the cache block chosen for replacement is invalidated and its active bit is set to 1 – it is worth recalling that this is done to prevent the deadlock scenario discussed earlier.
- Issue request to home-directory. Now, the write-miss is issued to its home directory. If the corresponding cache block is found to be uncached in any of the other processors, the directory replies with an empty write-list. If the cache block is indeed cached in some other processor, the directory replies with the list of pending write-misses being serviced in the directory. To minimize network traffic, before sending the write-list, a bloom filter is used to compress the list of addresses.
- Process response. Once the processor receives the write-list, if it indeed receives a non-empty write-list, it updates the local write-list register. When the write-miss receives all of its invalidation acknowledgements, it completes. When the write-miss completes, it sends a completion message to the directory, so that the directory knows about its completion. When a write-miss receives all of its acknowledgements and each of the acknowledgements are tagged inactive, its corresponding conflict-active bit (in the AWB) is reset to 0. When the conflict-active is reset, the write-miss checks the AWB to see if there are any entries prior to it. If there are none, it implies that the write-miss has become inactive and can be removed from the AWB.
- Remove write-miss and other inactive entries from the AWB. Before removing the write-miss entry, we first identify those memory operations which follow the original write-miss that have also become inactive. To identify such inactive memory operations, the AWB is scanned sequentially starting from the original entry, selecting all those entries whose conflict-active bit is reset to 0; the scanning is stopped when an entry whose conflict-active bit is set to 1 is encountered. All such selected entries are removed from the AWB, and the active bits of their respective cache blocks are reset to 0.

(Cache-hit actions). The actions performed when a cache-hit reaches the head of the ROB are as follows:

- AWB empty. The AWB is first checked to see if it is empty. If the AWB is empty, the cache-hit completes without checking for conflicts.
- Check for conflicts. If the AWB is not empty, the write-list registers are checked to see if the cache-hit conflicts with it. If the cache-hit conflicts (or if write-list register does not cache the directory entries of the cache-hit's home node), the cache-hit is treated like a miss and is issued to its home directory.
- No conflicts. If the cache-hit does not conflict, it is allowed to safely complete. Since the AWB is non-empty, the cache-hit is marked active by inserting it into the AWB with the write-miss bit set to 0, the conflict-active bit set to 0 (since it is a cache-hit, its predecessors in the conflict order must have completed), and the active bit of the cache block is set to 1.

(Read-miss actions). The actions performed when a read-miss reaches the head of the ROB are as follows:

- AWB empty. The AWB is first checked to see if it is empty. If the AWB is empty, the read-miss completes without checking for conflicts. Then, the data value reply that the read-miss received as a coherence response is examined. If it is tagged active, it is inserted into the AWB with the write-miss bit set to 0, the conflict-active bit set to 1 (since it obtained an active response), and the active bit of the cache block is set to 1. Later, when the read-miss receives its inactive response, the read-miss entry is removed along with subsequent inactive entries, as discussed earlier.
- Check for conflicts. If the AWB is not empty, the write-list registers are checked to see if the read-miss conflicts with it. If the read-miss conflicts (or if write-list register does not cache the directory entries of the read-miss' home node), the cache-miss is re-issued to its home directory.
- No conflicts. If the read-miss does not conflict, the read-miss is allowed to safely complete. Since the AWB is non-empty, the read-miss is marked active by inserting into the AWB with the write-miss bit set to 0. The conflict-active bit is set to 0 or 1 depending on whether the read-miss received an inactive or an active data value response, respectively. If it received an active response – later, when the read-miss receives the inactive response, the read-miss entry is removed along with subsequent inactive entries, as discussed earlier.

(Coherence and replacement requests). When a coherence request (invalidate or data value request) is received for a cache block that is marked active/inactive, the coherence response is in turn tagged active/inactive. Additionally, if the coherence request is for an active cache block, the coherence request is buffered in the req-buffer. When the cache block is eventually reset to inactive, the corresponding entry is removed from the req-buffer and the processor again responds to the buffered request – but now with an inactive response. When a coherence request is received for an active cache block and the req-buffer is full, the coherence response is delayed until a space in the req-buffer frees up. As discussed earlier, this cannot cause a deadlock, since active cache blocks will eventually turn inactive. Finally, a cache block that is marked active is not allowed to be replaced. When a replacement request is received, and all cache blocks in the set are marked active, the response is delayed until once of the blocks in the set turns inactive – again, without the risk of a deadlock.

5. Experimental Evaluation

We performed our experiments with several goals in mind. First and foremost, we want to evaluate the benefit of ensuring SC via conflict ordering in comparison with the baseline SC and RMO implementations. We then study the effect of varying the values of the parameters in our HW implementation, on the performance. Since the performance of our approach is dependent on how fast requests can get back replies from directories, we study the sensitivity towards the network latency. We also vary the size of write-lists to evaluate their effects on performance. We then study the characterization of conflict ordering to see how it reduces the overhead of using in-window speculation technique. Furthermore, we measure the additional network bandwidth that is used up and finally, we also measure the on-chip hardware resources that conflict ordering utilizes. However, before we present the results of our evaluation, we briefly describe our implementation.

5.1 Implementation

Processor	8, 16 and 32 core CMP, out of order
ROB size	176
L1 Cache	private 32 KB 4 way 2 cycle latency
L2 Cache	shared 8 MB 8 way 9 cycle latency
Memory	300 cycle latency
Coherence	directory based invalidate
# of AWB entries	50 per core
# of req-buffer entries	8 per core
# of write-list registers	6 per core
write-list size	160 bits
Interconnect	2D torus (2×4 for 8-core, 4×4 for 16-core, 4×8 for 32-core) 5 cycle hop latency

Table 1. Architectural Parameters.

Benchmark	Inputs
barnes	16K particles
fmm	16K particles
ocean	258× 258
radiosity	batch
raytrace	car
water-ns	512 molecules
water-sp	512 molecules
cholesky	tk15.O
fft	64K points
lu	512×512
radix	1M integers

Table 2. Benchmarks.

We implemented conflict ordering using SESC [24] simulator, targeting the MIPS architecture. The simulator is a cycle-accurate, execution-driven multi-core simulator with detailed models for the processor and the memory systems. To implement conflict ordering, we added the associated control logic to the simulator. We considered conflict ordering in the context of a CMP with local caches kept coherent using a distributed directory based protocol. The architectural parameters for our implementation are presented in Table 1. The default architectural parameters were used in all experiments unless explicitly stated otherwise. We measured performance with 8, 16 and 32 processors in Section 5.2, and for other studies, we assumed 32 processors. We used the SPLASH-2 [32], a standard multithreaded suite of benchmarks for our evaluation. We could not get the program *volrend* to compile using the compiler infrastructure that targets the simulator and hence we omitted it. We used the input data sets described in Table 2 and ran the benchmarks to completion.

(Our baseline SC implementation). Our SC baseline, referred to as conventional SC in the experiments below, uses in-window speculation support. It is an aggressive implementation that uses hardware prefetching and support for speculation in modern processors to speculatively reorder memory operations while guaranteeing SC

ordering using replay. However, such speculation is within the instruction window – where the instruction window refers to the set of instructions that are in-flight. The implementation we use is similar to ones used as SC baselines in proposals such as [6, 14, 31].

(Our baseline RMO implementation). Our baseline RMO implementation allows memory operations to be freely reordered, enforcing memory ordering only when fences are encountered. Even when fences are encountered, in-window speculation support (as used in the SC baseline) is used to mitigate the delays. The RMO implementation is similar to ones used in recent works such as [6, 14, 31].

5.2 Execution time overhead

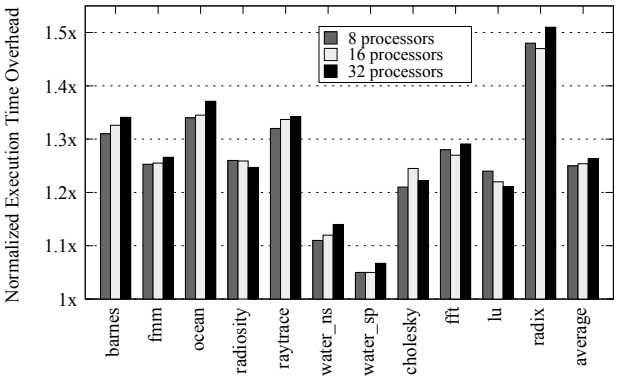

Figure 8. Conventional SC normalized to RMO

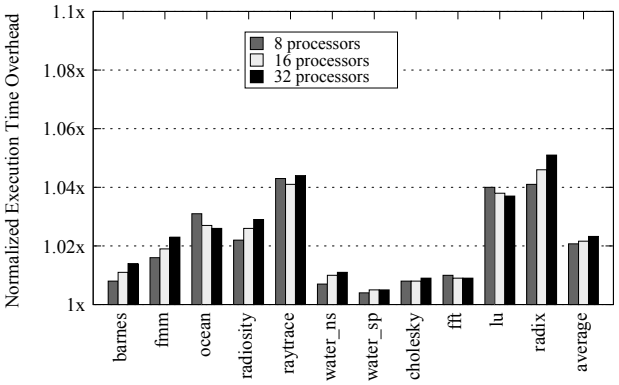

Figure 9. Conflict ordering normalized to RMO

We measure the execution time overhead of ensuring SC via our approach conflict ordering and compare it with the corresponding overhead for conventional SC. We conducted this experiment for 8, 16 and 32 processors using the default hardware implementation presented in Table 1. Fig. 8 and Fig. 9 show the execution time overheads for conventional SC and conflict ordering, respectively. The execution time overheads are normalized to the performance achieved using RMO. As we can see, most benchmark programs experience significant slowdown for conventional SC (more than 20% overhead on average for all numbers of processors). In particular, *radix* has the highest overhead. This is because *radix* has a relatively high store-miss rate which forces the following loads to wait longer before they can be retired. As we can see, with conflict ordering the overhead of ensuring SC is significantly reduced. On average, the overhead is just 2.0% for 8 processors, 2.2% for 16 processors and 2.3% for 32 processors, and thus the performance of our approach is comparable to RMO. With conflict ordering loads and stores do not need to wait until outstanding stores complete; they can mostly retire as soon as the pending stores get replies back from directories. Since the time to get replies from directories is significantly less than the time for outstanding stores to complete, loads do not end up causing stalls and stores can also be performed out of order. This explains why the performance with conflict ordering is significantly better than conventional SC. Furthermore, it is worth noting that, with different numbers of processors, the performance does not vary significantly. Therefore, our approach appears scalable.

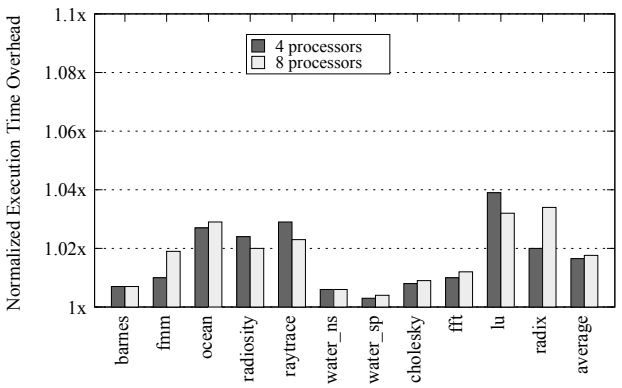

Figure 10. Performance for bus-based implementation.

(Bus-based implementation). In addition to the distributed directory-based implementation, we also evaluated a bus-based implementation to find out if conflict ordering is applicable to current multi-core processors which are predominantly bus based. For our bus based implementation, we just implement basic conflict ordering; we maintain a centralized on-chip structure called *write-list-buffer* (WLB), which records the addresses of the write-misses which are currently pending. When a write-miss retires into a local write-buffer it is sent to the WLB; upon receiving the write-miss, the WLB inserts its address into the WLB and replies back with the write-list – containing all the addresses that are currently present in the WLB except the addresses from the source processor. Similar to the directory-based implementation, a bloom filter is used to compress the addresses. Memory operations which attempt to complete past the write-miss, check the received write-list to decide whether they can be completed safely, without violating SC. We conducted experiments for bus-based implementation with 4 and 8 processors, and set the round-trip latency for accessing WLB as 5 cycles. Fig. 10 shows the execution time normalized to RMO. As we can see, the execution time of our technique is close to RMO for both 4 and 8 processors, with the overhead less than 2% on average. This shows that conflict ordering is also applicable to small scale multi-core processors that use a bus for coherence.

5.3 Sensitivity studies

(Sensitivity towards network latency). The performance of conflict ordering hinges on the time for a miss to get replies from the directory. It is desirable that conflict ordering is reasonably tolerant to the network latency. In our experiments, we used 2D torus network and varied the latency of each hop with values of 3 cycles,

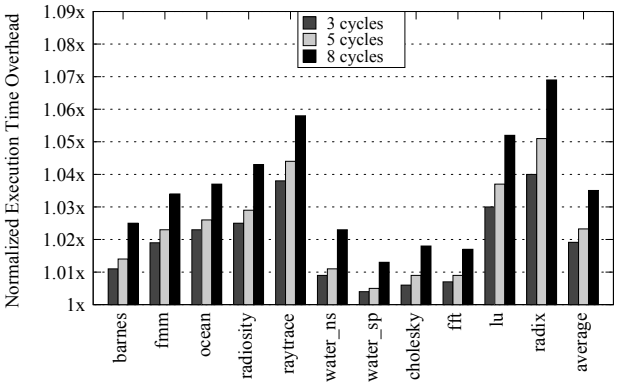

Figure 11. Varying the latency of network.

5 cycles, and 8 cycles, as shown in Fig. 11. The performance is measured with 32 processors. As we can see, the overhead does not vary significantly when the latency is increased from 3 cycles to 5 cycles. Even when the latency is increased to 8 cycles, the average performance drops only slightly, to 3.5%. This shows conflict ordering is reasonably tolerant to the network latency. In our default design, we choose 5 cycles as each hop latency, assuming 2 cycle wire delay between routers and 3 cycle delay per pipelined router.

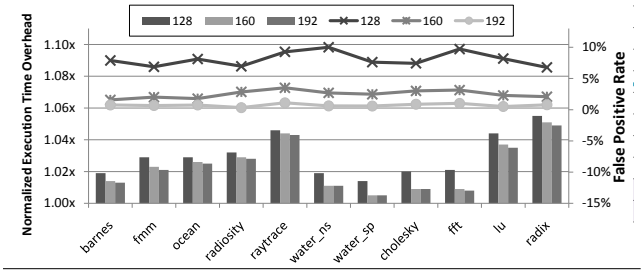

Figure 12. Varying number of bits of write-list.

(Sensitivity towards size of write-list). Recall that, to minimize network bandwidth, the addresses within replies to cache miss requests are compressed to form a write-list using a bloom filter. While a smaller size is beneficial as far as saving network bandwidth is concerned, it could also result in false positives. A false positive can lead us to falsely conclude that a load or store conflicts with a pending store, causing the load to re-execute or the store to stall. In this experiment, we evaluated the minimum size of the write-list that does not result in performance loss. We varied the number of bits of write-list with the value 128, 160, and 192 to evaluate the performance and corresponding false positive rates with 32 processors. In our implementation, we used a bloom filter with 4 hash functions. Fig. 12 shows the results, where lines represent false positive rates and bars represent execution time overheads. As the number of bits increases from 128 to 192, the false positive rate decreases (on average, 8.01%, 2.42%, and 0.69% respectively), and the execution overhead also decreases (on average, 2.97%, 2.33% and 2.29% respectively). A size of 160 bits performs slightly better than 128 bits and very close to 192 bits. Therefore, we choose 160 bits (20 bytes) as the size of the write-list in our implementation.

5.4 Characterization of conflict ordering

In this experiment, we wanted to examine how conflict ordering reduces the overhead of using conventional SC. Recall that, in the conventional SC implementation, loads cannot be retired if there are pending stores and stores cannot be performed out of order, which leads to the gap between the performance of SC and RMO. On the other hand, using conflict ordering, we can safely retire loads and complete stores past pending stores by checking the write-list. Hence, performance hinges on how often loads and stores can be reordered with their prior pending stores.

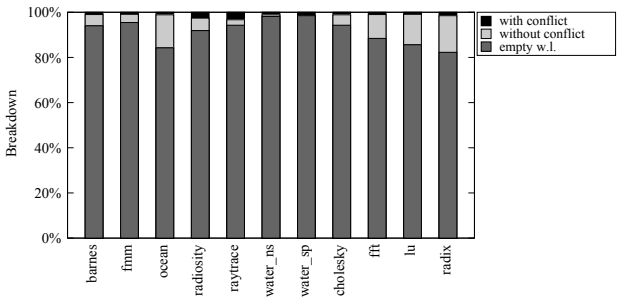

Figure 13. Breakdown of checks against write-lists.

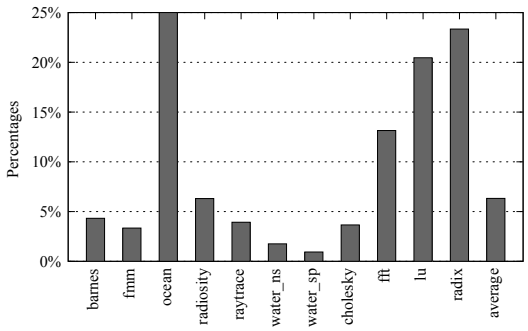

Figure 14. Reduced accesses to directories.

Fig. 13 shows the breakdown for all checks against write-lists. We categorize these checks into three types: checks against empty write-lists, checks against non-empty write-lists without finding any conflict, and checks against non-empty write-lists with finding a conflict. When a request to the directory finds that the targeted block is not cached, it indicates that no active memory operation has accessed the block and the requesting processor can get a reply with an empty write-list. As we can see, most checks are against empty write-lists (around 90% on average). For the checks against non-empty write-lists, there is almost no conflict. Hence, our approach allows almost all memory operation to be reordered, achieving performance comparable to RMO.

Fig. 14 shows the reduced directory accesses using enhanced conflict ordering, compared to basic conflict ordering. The performance of our approach also hinges on the frequency of directory accesses. Hence, it is important to have fewer directory accesses. As we can see, using enhanced conflict ordering, on average we only have about 6% directory accesses of basic conflict ordering. This is because, for most benchmark programs, AWB is empty most of the time. For some benchmarks, such as *ocean*, the access frequency does not reduce as much as other benchmarks. However, the access frequency for these benchmarks in basic conflict ordering is already low. Therefore, their overhead is still low.

5.5 Bandwidth increase

In this experiment, we measure the bandwidth increase due to write-lists that need to be transferred and extra traffic introduced by

Benchmark	Bandwidth increase (%)
barnes	2.49
fmm	0.98
ocean	2.64
radiosity	8.36
raytrace	1.21
water-ns	2.08
water-sp	2.21
cholesky	0.24
fft	1.16
lu	1.96
radix	1.71

Table 3. Bandwidth Increase.

conflict ordering for 32 processors. Table 3 shows the bandwidth increase compared to RMO. As we can see, for most benchmarks, the bandwidth increase is less than 3% (2.27% on average). *radiosity* has relatively higher bandwidth increase. This is because its write miss rate is relatively higher and requests to directories usually get non-empty write-lists, resulting in write-lists taking up a relatively high proportion of bandwidth.

5.6 HW resources utilized

Recall that the additional HW resources utilized by conflict ordering are AWB, req-buffers, write-list registers, and the active bits added to each cache block. Each AWB entry contains the cache block address, the conflict-active and the write-miss bits; we do not count the storage required for data written by the write-miss, since it is already present in conventional write buffers. Since we use 50 AWB entries per core, each of size 5 bytes, the total size per processor core amounts to 250 bytes for the AWB. Since we use 8 entries for the req-buffer, each of size 6 bytes (for storing the cache block address and the processor id), the total size per processor core amounts to 48 bytes. With 6 write-list registers per core, each of size 20 bytes, the total size per processor core amounts to 120 bytes. In addition, we also require active bits to be added to each cache block in the L1 cache. This amounts to an additional 64 bytes of on-chip storage per processor core. Thus the total additional on-chip storage space amounts to 482 bytes per processor core. In addition to this we require the hardware resources needed for in-window speculation which is also required by our SC and RMO baselines. Thus, the additional hardware resources utilized for conflict ordering is nominal.

6. Conclusion

Should hardware enforce SC? Researchers have examined this question for over 30 years, with no definite answers, yet. While there has been no clear consensus on whether hardware should support SC [3, 16], it is important to note, however, that the benefits of supporting SC are widely acknowledged [3]. Indeed, critics of hardware enforced SC, question it based on whether the costs of supporting SC justify its benefits – all prior SC implementations needing to employ aggressive speculation and its associated complexity for supporting SC.

In this paper we demonstrate that the benefits of SC can indeed be realized using nominal hardware resources. While prior SC implementations guarantee SC by explicitly completing memory operations within a processor in program order, we guarantee SC by completing conflicting memory operations, within and across processors, in an order that is consistent with the program order. More specifically, we identify those shared memory dependencies whose ordering is critical for the maintenance of SC and intentionally order them. This allows us to non-speculatively complete memory operations past pending writes and thus reduce the number of stalls due to memory ordering. Our experiments with SPLASH-2 suite showed that SC can be achieved efficiently, incurring only 2.3% additional overhead compared to RMO.

Acknowledgments

We would like to thank the reviewers and our shepherd for their helpful comments and advice for improving this paper. This work is supported by NSF grants CCF-0963996 and CCF-0905509 to the University of California, Riverside, and by the Centre for Numerical Algorithms and Intelligent Software, funded by EPSRC grant EP/G036136/1 and the Scottish Funding Council to the University of Edinburgh.

References

[1] Intel Single-Chip Cloud Computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`.

[2] Telera Tile-Gx processor. `http://www.tilera.com/products/processors`.

[3] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29:66–76, 1995.

[5] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14, 1990.

[6] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 233–244, 2009.

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, 2007.

[8] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 97–108, 2007.

[9] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, 2007.

[10] X. Fang, J. Lee, and S. P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 285–294, 2003.

[11] M. Galluzzi, E. Vallejo, A. Cristal, F. Vallejo, R. Beivide, P. Stenström, J. E. Smith, and M. Valero. Implicit transactional memory in kilo-instruction multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, pages 339–353, 2007.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*, ISCA '91, pages 355–364, 1991.

[13] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 179–188, 2002.

[14] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 162–171, 1999.

[15] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, page 102, 2004.

[16] M. D. Hill. Multiprocessors should support simple memory-consistency models. *Computer*, 31(8):28–34, 1998.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[18] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50(8):824–833, 2001.

[19] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 295–306, 2010.

[20] 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 *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 210–221, 2010.

[21] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: a simple and efficient memory model for concurrent programming languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 351–362, 2010.

[22] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A case for an SC-preserving compiler. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 199–210, 2011.

[23] 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 *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 199–210, 1997.

[24] 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.

[25] K. G. Sarita, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. *Technical Report CSL-TR-93-594, Stanford University*, 1993.

[26] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ISCA '87, pages 234–243, 1987.

[27] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, 2008.

[28] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communication of the ACM*, 53(7):89–97, 2010.

[29] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, 1988.

[30] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient processor support for DRFx, a memory model with exceptions. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 53–66, 2011.

[31] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 266–277, 2007.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, 1995.

[33] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.