OmniOrder: Directory-Based Conflict Serialization of Transactions *

Xuehai Qian †
University of California, Berkeley
xuehaiq@berkeley.edu

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

Josep Torrellas
University of Illinois, Urbana-Champaign
torrellas@cs.uiuc.edu

http://iacoma.cs.uiuc.edu

Abstract

Effective execution of atomic blocks of instructions (also called transactions) can enhance the performance and programmability of multiprocessors. Atomic blocks can be demarcated in software as in Transactional Memory (TM) or dynamically generated by the hardware as in aggressive implementations of strict memory consistency. In most current designs, when two atomic blocks conflict, one is squashed — a performance loss that is often unnecessary.

To avoid this waste, this paper presents OmniOrder*, the first design that efficiently executes conflicting atomic blocks concurrently in a directory-based coherence environment. The idea is to keep only non-speculative data in the caches and, when the cache coherence protocol transfers a line, include in the message the history of speculative updates to the line. The coherence protocol transitions are unmodified. We evaluate OmniOrder with 64-core simulations. In a TM environment, OmniOrder reduces the execution time of the STAMP applications by an average of 18.4% over a scheme that squashes on conflict. In an environment with SC enforcement with speculation, we run 11 programs that implement concurrent algorithms. OmniOrder reduces the programs' execution time by an average of 15.3% relative to a scheme that squashes on conflict. Finally, OmniOrder's communication overhead of transferring the history of speculative updates is negligible.*

1. Introduction

To improve the performance and programmability of multiprocessors, there is significant interest in efficiently supporting the execution of atomic blocks of instructions — also called transactions. Atomic blocks can be demarcated in software by the programmer or the compiler, or generated dynamically by the hardware on ordinary code. Examples of the former include transactions in Transactional Memory (TM) [15] and code regions demarcated to enable aggressive compiler optimization [2]. Hardware-generated blocks are created, for example, in out-of-window speculation to prevent processor stall in aggressive implementations of strict memory consistency [6, 24].

Efficient atomic block execution requires system support for a handful of architectural mechanisms. They include fast register checkpointing, speculative execution of an atomic block with temporary state buffering, detection of data conflicts (i.e., data dependences) between a block and code executing on another processor, and atomic block squash and commit. Block squash involves discarding its state and rolling back to the checkpoint; block commit involves making the state generated by the atomic block irreversibly visible to the rest of the system. Computer manufacturers such as IBM, Intel, AMD, Oracle, and Azul have introduced or plan to introduce some of these mechanisms in their systems.

Most of the proposed designs squash an atomic block on a conflict with code executed on another processor — i.e., one processor reads and the other writes to the same line, in either order, or both write. Unfortunately, squashing blocks has a cost in performance and energy, which can be substantial if squashing is frequent. A few proposals stall the block until it is safe to continue, but this still costs performance.

In practice, in many conflicts between two atomic blocks, there is no need to squash or stall. This is the case, for example, when the source of the dependence (i.e., the predecessor processor) and the destination (i.e., the successor) access different parts of the same memory line. Moreover, even if they access the same address, a squash can be avoided by eventually committing the predecessor block first. In all of these cases, the conflicting atomic blocks can execute concurrently without squashes, and produce a state as if they had executed serially. In other words, they are conflict serializable.

Supporting conflict serialization of atomic blocks can improve performance, but introduces complexity. First, the system has to keep a record of the conflicting blocks and their predecessor-successor order — likely forming, by transitivity, a chain of successor blocks. The type of dependence, such as anti, output, true, and false, may also be recorded. Second, the system needs to detect dependence cycles, where one block ends up depending on itself through other blocks. In this case, since the execution cannot be serialized, at least one of the blocks needs to be squashed. Third, when a block is squashed, we also need to squash at least all of its successors that have true dependences (same-address RAW) with it — and recursively, these successors' successors. Finally, the commit of dependent blocks has to be done in strict order.

The goal of this paper is to efficiently support conflict serialization of atomic blocks in a distributed directory-based coherence environment. Currently, there are several proposals that support conflict serialization of atomic blocks, most notably DATM [20], SONTM [4], BulkSMT [18], and Wait-n-GoTM [16]. However, they all fall short of the goal of this paper. Specifically, DATM is designed for a broadcast-based protocol. SONTM can be used in a directory setting, but it adds many additional messages during block execution and commit.

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

†Xuehai Qian did this work while at the University of Illinois.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

BulkSMT is designed for SMT processors. Finally, Wait-n-GoTM uses a hardware-software design that makes it hard to apply to atomic blocks generated dynamically to support an aggressive implementation of strict memory consistency. Moreover, on a block squash, all of the block's successors get squashed, irrespective of whether or not they have true dependences with it.

In this paper, we present *OmniOrder*, the first architecture that efficiently executes conflicting atomic blocks concurrently in a directory-based coherence environment. The blocks appear to be serialized. OmniOrder uses eager conflict detection and can be used with both software-demarcated and hardware-generated atomic blocks. The main idea behind OmniOrder is to keep only non-speculative data in the caches and, when the cache coherence protocol transfers a line, include in the message the history of speculative updates to the line. The coherence protocol transitions are unmodified, and are decoupled from the management of the speculative data. When an atomic block commits, its updates are merged into the caches, wherever they are. When a block is squashed, its updates are discarded, also wherever they are.

Another key attribute of OmniOrder is its efficient state recovery. Specifically, on a block squash, OmniOrder only squashes the block's successors that have same-address RAW dependences with the block (and recursively, these successors' successors). Also, OmniOrder has no centralized structure.

We evaluate OmniOrder in the context of both TM and Sequential Consistency (SC) enforcement via out-of-window speculation. We simulate a 64-core manycore. For TM, we compare OmniOrder to InvisiFence [6] which, on a conflict between two atomic blocks, squashes one of them. We show that OmniOrder reduces the execution time of the STAMP applications by an average of 18.4%. For SC enforcement, we compare OmniOrder to five other speculative schemes that do not allow concurrent execution of dependent atomic blocks. We find that, on average, SPLASH-2 and PARSEC applications have too few conflicting blocks for OmniOrder to make a difference. However, we also run 11 programs that implement concurrent algorithms such as non-blocking queues, and that induce frequent speculation. In this case, OmniOrder reduces the execution time by an average of 15.3% relative to InvisiFence with Commit-On-Violation. Finally, we show that the communication overhead of transferring the history of speculative updates in OmniOrder is negligible.

This paper is organized as follows: Section 2 provides a background; Sections 3 and 4 present the design and implementation of OmniOrder in the context of TM; Section 5 discusses enforcing strict memory consistency; Section 6 evaluates OmniOrder; and Section 7 discusses related work.

2. Executing Conflicting Atomic Blocks

2.1. Existing Proposals

There are a few proposals that support conflict serialization of atomic blocks, such as DATM [20], SONTM [4], BulkSMT [18], and Wait-n-GoTM [16]. While each of these schemes advances the state-of-the-art in some directions, none provides an effective approach usable in a distributed directory-based protocol for both software-demarcated atomic blocks (e.g., in TM) and hardware-initiated blocks (e.g., in speculation for strict memory consistency).

DATM [20] uses cache-coherence protocol transactions to detect dependences between atomic blocks. Then, it records the order of the blocks in a centralized order vector. The order vector is used to select the correct data version needed on an access, to order the commits, and to detect dependence cycles. An enhanced snoopy-based cache coherence protocol supports the forwarding of speculative updates between caches like an update-based protocol. Overall, this design is not suitable for a directory scheme because updating the global order vector needs broadcasts. Moreover, DATM couples the management of the speculative data with the coherence protocol, adding 11 stable states to the protocol.

SONTM [4] orders dependent atomic blocks using Serializability Order Numbers (SON). Each block has an upper-bound and a lower-bound SON. Each memory location accessed has a read-number and a write-number stored in memory. While some optimizations are possible, each load and store instruction in a block needs to get this read-number or write-number to potentially update the block's upper and lower bounds. If the upper bound ends up being smaller than the lower bound, then the block cannot be serialized with other conflicting blocks and it is squashed. At block commit, a validation step involves broadcasting write-numbers of all the updated data and receiving read-number responses from other processors. Overall, while SONTM can be applied to a directory scheme, it adds substantial execution overhead.

BulkSMT [18] enhances a Simultaneous Multithreaded (SMT) processor to support the execution of conflicting atomic blocks. It detects dependences between blocks using cache access bits, and records block orders using simple, centralized hardware tables. This design is not applicable to directories.

Wait-n-GoTM [16] is based on TokenTM [8], and uses a hardware-software combination to execute conflicting atomic blocks concurrently. Specifically, cross-block conflicts are detected with cache-coherence protocol transactions. In the sucessor processor, the hardware logs the ID of the predecessor processor in a log software structure. Cycles are detected with two hardware timestamps per atomic block that are updated during communications. On a block squash, a software handler restores the state and traverses the log software structure to identify all the successors, squashing them all — irrespective of the type of data dependence present. When a block with conflicts wants to commit, a software handler decides whether it can commit and which other blocks can now commit as well. Wait-n-GoTM is applicable to directory-based coherence. However the protocol requires software support for dependent-block commits and squashes, making it applicable to TM, but not to speculation in aggressive implementations of strict memory consistency models.

2.2. Design Guidelines

We propose some design guidelines to support conflict serialization of atomic blocks in directory-based systems.

• **Decouple the speculative state of data from the coherence protocol state.** Design complexity is minimized if the state transitions of the cache-coherence protocol are unaffected by whether or not the data is speculative. This guideline is followed by SONTM and Wait-n-GoTM, but not by DATM.

• **Support efficient state recovery.** For high performance, the system should efficiently recover when an atomic block (B) is squashed. Specifically, in addition to discarding B's state, the system should access the chain of B's successors and discard only the state generated by those that have *true* (i.e., same-address RAW) data dependences with it (recursively).

• **Separate atomic-block ordering from data-version granularity.** Conflicting atomic blocks should be ordered in a successor chain based on dependences at the granularity of cache *lines* — since this is the granularity detected by the coherence protocol. However, to be able to squash only blocks with true data dependences, the system should record data versions at the granularity of the *accesses* (i.e., words or bytes). Otherwise, unnecessary squashes occur.

• **Minimize overhead if there are no conflicts.** The execution of atomic blocks that do not have conflicts should add minimal overhead to the system — e.g., it should not incur additional memory accesses to update time stamps. SONTM suffers from this overhead.

• **Have no centralized structure.** To operate in a scalable distributed-directory environment, the design cannot have any centralized hardware structures, as they become bottlenecks.

3. OmniOrder Design

In the context of a directory protocol, OmniOrder provides efficient conflict serialization of atomic blocks for many types of conflicts. Atomic blocks can be demarcated in software, as under TM [15] or for compiler optimization [2]. They can also be generated dynamically by the hardware, e.g., to avoid stalls due to strict memory consistency models [6, 24]. In this section, we focus on a TM system; we consider speculation for memory consistency later. Without loss of generality, our discussion assumes an MSI invalidation-based distributed directory scheme. Each processor has a private L1 cache and shares the L2 cache with all the other processors. We start by presenting the main idea, and then describe the coherence operations.

3.1. Main Idea

The main idea behind OmniOrder is to decouple speculative state management from the coherence protocol state transitions. Even though processors execute transactions speculatively, and speculative state from one transaction propagates to other transactions in other processors, we do not modify the basic cache coherence protocol. This is accomplished by recording, for each line, the current history of speculative updates to it, and keeping this history in a small per-processor buffer. The history is then piggybacked on the coherence transactions for the line. Cache lines always contain the *non-speculative* version of the data, and only transition between ordinary coherence states. This approach keeps the hardware simple.

For example, consider a memory line that is updated by two transactions in two different processors, namely P0 and P1. The first update brings the non-speculative version of the line in Modified (M) state to P0's cache, and the speculative update is stored in a buffer in P0. The second update moves the line from P0's cache to P1's cache, sets it to M state in P1, and invalidates the entry from P0's cache – exactly as in the ordinary protocol. The speculative update by P0 is piggybacked on the transaction and removed from P0's buffer. It is appended to the speculative update by P1 and stored in a P1 buffer. P1 is now responsible for both speculative updates, since P0 has lost all such information. Later, when P0's transaction commits, it will send a signal to its successor P1, which will trigger the *merge* of P0's speculative update into the line in P1's cache. If, instead, P0's transaction is squashed, P0 sends a signal to P1, which prompts the removal of P0's speculative update. In addition, since P0-P1 have a WAW dependence, there is no need to squash P1's transaction.

In its operation, OmniOrder links conflicting transactions with predeccessor/successor pointers. Specifically, every time that a cache line shared by transactions suffers a cache coherence transition, the source transaction(s) become(s) the *Predecessor(s)* and the destination transaction becomes the *Successor*. For example, Figure 1 shows a trace of accesses by four transaction-executing processors (P0-P3) to same memory line. First, P1 writes (w_{P1}) and the line is loaded in M state. Then, P0 writes, and both P0 and P1 change state. Hence, P1 and P0 have a predecessor-successor relationship (P1→P0). Then, P2 and P3 read the line, and P0, P2, and P3 transition to Shared (S) state. The two pairs (P0→P2 and P0→P3) have predecessor-successor relationships. Finally, P3 writes to the line, and so we have the relationship P0→P3 and P2→P3. Note that, in an MSI protocol, a processor read-missing on a line in state S elsewhere gets the line directly from the shared L2 cache — rather than from another L1 cache. Consequently, in our producer-consumer links, we will sometimes have to link-in the directories as well (Section 3.2.1).

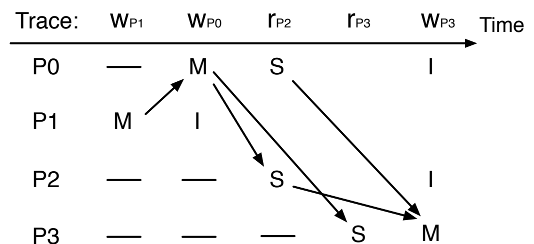

Figure 1: Example of a trace of accesses by four processors to the same cache line. The table shows line states.

OmniOrder uses the predecessor-successor links to propagate information about commits and squashes. For example,

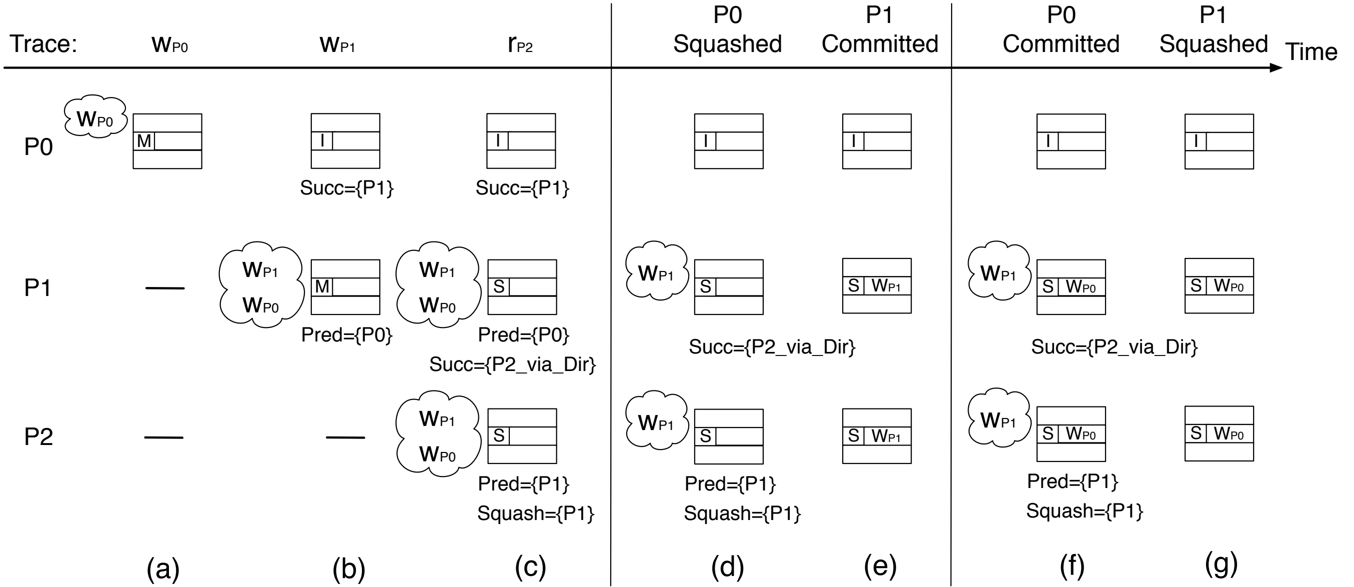

Figure 2: Example of OmniOrder's operation. All accesses are to the same word.

when P1 commits (or gets squashed), the hardware informs P0, which in turn informs P2 and P3, and so on. However, these links do not automatically trigger squash operations. Indeed, when a transaction P is squashed, its chain of successors is followed, but only those transactions S that have *RAW* dependences to the *same* address (word or byte depending on the finest granularity of access information kept) with P are squashed. For example, in Figure 1, assume that w_{P1} updates word a, w_{P0} and r_{P2} access word b, and r_{P3} reads word a — where a and b share the same line. Then, if the transaction in P1 gets squashed, as the information is propagated down the chain of successors, P0 and P2 are not squashed because they accessed a different word. Only P3 is squashed. OmniOrder's support for this model saves squashes.

3.1.1. Example. To illustrate OmniOrder's operation, Figure 2 shows an example of three transaction-executing processors accessing the same word. First, in Chart (a), P0 writes to the word (w_{P0}), which brings the original line to its cache in state M. However, the speculative update is kept separately (shown as a cloud). In Chart (b), P1 has written to the word (w_{P1}). As a result, P0's cache line and speculative buffer have been invalidated, while P1 loads the unmodified line in state M and takes responsibility of the two (ordered) speculative updates in its speculative buffer. In addition, P0 and P1 have been linked as predecessor-successor. In Chart (c), P2 has read the word (r_{P2}). Hence, both P1 and P2 caches now keep the (unmodified) line in state S, and both keep the two speculative updates in their buffers. In addition, P1 and P2 have linked as predecessor-successor. In Chart (c), we write that P1's Successor set is "P2_via_Dir". The reason is that, because future readers of the line will get the line from the L2 cache without informing P1, we need to keep some information in the directory. We will explain the design later. In addition, P2 has created a *Squash* set that includes P1. It means that, if P1 is squashed, then P2 needs to be squashed as well. The set does not include P0 because P1's write isolates P2 from P0. In addition, P1 does not have a Squash set because it has a WAW with P0.

Assume now that P0 gets squashed and then P1 commits. When P0 gets squashed (Chart (d)), a signal is propagated through the complete P0 successor chain. In each of these processors, starting from P0, P0's speculative updates in the buffers are discarded, and P0 is removed from any Predecessor set. In this special case, where the squashed processor has no predecessors, its Successor set is also cleared. Later, when P1 commits (Chart (e)), a signal is propagated through P1's successor chain. In each of these processors, starting from P1, P1's speculative updates in the buffers are merged into the cache line, and P1 is removed from any Predecessor and Squash set (we will see that the L2 is automatically updated). The successor set of the committing processor is cleared. Note that squashes and commits do not change coherence states.

Assume that, instead of (d) and (e), P0 commits (Chart (f)) and P1 then gets squashed (Chart (g)). When P0 commits (Chart (f)), the signal through P0's successor chain triggers the merge of P0's speculative updates into the caches, and the removal of P0 from any Predecessor set. P0's Successor set is cleared. Later, when P1 gets squashed (Chart (g)), a signal through P1's successor chain discards P1's speculative updates from all buffers (including P1's) and clears P1 from any Predecessor sets. In addition, any processor that has P1 in its Squash set gets squashed. This happens to P2. As usual, P1's Successor set is cleared and P1 is removed from P2's Squash set. The caches of the squashed processors (P1 and P2) keep their whole state, which was non-speculative.

3.2. OmniOrder's Basic Operation

We now describe OmniOrder's basic operation. The speculative buffer in each processor is divided into two structures. First, the speculative updates by the *local* processor are stored

in a cache called *L0*, which is accessed before the main cache (L1). This ensures that locally-produced data can be accessed quickly. Secondly, the history of speculative updates by *predecessor* transactions are stored in a buffer called *Speculative Version Buffer (SVB)*. Each SVB entry contains the history of updates for one word (since we assume here that words are the finest granularity tracked). An entry contains an address and an ordered list of tuples containing the writer processor ID and the data. The SVB is accessed in parallel with L1. On an external access, the relevant L0 and SVB entries are read and combined to form the complete history, which is provided to the requester. A cache line with state in L0 or SVB cannot be displaced from L1, even if it is in state S. Before doing so, it triggers a transaction squash (Section 4.2).

Each processor has a *Predecessor Set* (*Pred_Set*) and a *Successor Set* (*Succ_Set*) bitmask. Each contains as many bits as processors. OmniOrder sets bit i of *Pred_Set* or *Succ_Set* if processor i is found to be a predecessor or a successor, respectively, of the local processor. Each processor also has a *Squash Set* (*Sq_Set*) bitmask. Bit i is set if the squash of a transaction in processor i induces the squash of the local transaction. Recall that this only occurs for RAW dependences on the same word. Finally, as we will see next, the directory can sometimes act as the "successor" of a processor in the dependence chain. Consequently, each processor has a *Successor Directory Set* (*Succ_Dir*) bitmask. It contains as many bits as directory modules. OmniOrder sets bit i if directory module i acts as a successor of the local processor.

OmniOrder requires that, on a coherence operation, the source and destination processors know and record each other's ID. This is easy in directory-based schemes, where messages are acknowledged and include the sender's ID.

3.2.1. Role of the Directory in Maintaining the Predecessor-Successor Chain. In an ordinary MSI directory protocol, when a private cache read-misses on a line that is in M state in another private cache, the line ends up being written back to the shared cache (e.g., L2). Future read misses will be satisfied from the shared cache; not from the last writer's cache. Hence, in OmniOrder, the directory in the shared L2 cache must participate in building the predecessor-successor chain.

Consider a producer processor (P_{prod}) that caches a line in M state with a history of speculative updates. In OmniOrder, when a consumer processor (P_{cons}) read misses on the line, as the history of speculative updates is sent to P_{cons}, the corresponding directory module in L2 (D_i) also keeps a copy of the history. From then on, D_i acts as P_{prod}'s *proxy* for this line. Specifically, while P_{cons} adds P_{prod} to its *Pred_Set*, P_{prod} does not update its *Succ_Set*. Instead, P_{prod} sets the bit in its *Succ_Dir* corresponding to D_i, making D_i a successor. Moreover, each directory module also has as many *Succ_Set* bitmasks as the number of processors in the machine. Consequently, D_i sets the bit for P_{cons} in its *Succ_Set* bitmask that corresponds to P_{prod}.

From now on, future read misses to the line by other processors, as they reach directory D_i, will be satisfied by D_i. D_i will respond with the memory line, the history of speculative updates, and the ID of the last writer (P_{prod}). D_i will then set the *Succ_Set* bit corresponding to the new consumer. The latter will set the *Pred_Set* bit corresponding to P_{prod}.

With this design, the directory ensures the continuity of the predecessor-successor chain. When P_{prod} sends commit or squash signals, it sends them to both the processors in its *Succ_Set* and the directory modules in its *Succ_Dir*. The directory modules will forward them to the processors in their own *Succ_Set* for P_{prod}. Note that, with this design, it is possible that a processor receives more than one message for the same commit or squash event in another processor. OmniOrder is designed to handle this case.

In summary, each directory module contains an SVB buffer like that in the processors, with entries coming from the SVBs of multiple processors. In addition, it has an array of as many *Succ_Set* bitmasks as there are processors in the machine.

3.2.2. Read Access in a Transaction. Figure 3(a) outlines the algorithm followed when a processor (P_r or requesting processor) issues a request inside a transaction that is satisfied by a second processor (P_s or supplier processor) also executing a transaction. The steps *may* involve updating: (i) the coherence state of the line in the P_r and P_s caches, (ii) L0 and SVB in P_r and P_s, (iii) *Succ_Set* and *Succ_Dir* in P_s, (iv) *Pred_Set* and *Sq_Set* in P_r, and (v) *Succ_Set* and SVB in one directory module. In this section, we describe the case of a read.

Transactional Read/Write by Pr that is Satisfied by Ps

Depending on the coherence operation, update:
1. Coherence state of the line in the Pr and Ps caches
2. L0 and SVB in Pr and Ps
3. Succ_Set and Succ_Dir in Ps
4. Pred_Set and Sq_Set in Pr
5. Succ_Set and SVB in one directory module

(a)

Transaction Commit in P

1. Merge all the speculative updates by P (locally and remotely)
2. Tell P's successor chain of P's commit [use Succ_Set and Succ_Dir]
   2.1. Transactions in the successor chain may start commits
3. Clear the local speculative hardware structures

(b)

Transaction Squash in P

1. Discard all the speculative updates by P (locally and remotely)
2. Tell P's successor chain of P's squash [use Succ_Set and Succ_Dir]
   2.1. Transactions in the successor chain may start squashes
3. When all the earlier transactions are committed or squashed, clear the local speculative hardware structures and restart

(c)

Figure 3: OmniOrder algorithms for a read/write in a transaction (a), transaction commit (b) and squash (c).

Consider first a read miss. The non-speculative version of the line is brought in S state to P_r's cache. If there is a history of speculative updates, it is piggybacked on the same message and stored in P_r's SVB. The last writer of the line (irrespective of the word it wrote) is the predecessor. It will be the one that will forward commits and squashes P_r.

OmniOrder needs to update the state of the directory since, from now on, it will have the role of providing the line. The actions taken depend on whether the line was in M state in the last writer's cache or in S state in multiple caches. In the first case, the request reaches P_s, which caches the line in M state. P_s changes the line to S state and creates the line's update history by combining its L0 entries with its SVB entries (although L0 and SVB are unmodified). Then, it sends the line and history to the directory module that maps the line (say *i*). It also sets the *i* bit in its *Succ_Dir* bitmask. Directory module *i* is now P_s's successor and proxy for this line. Hence, it saves the line in L2, copies the history to its SVB, and sets the P_r bit in its *Succ_Set* for P_s. Then, it forwards line, history, and P_s's ID to P_r.

If, instead, the line was in S state in multiple caches, the response comes directly from the directory module that maps the line. That directory knows P_s's ID from the last writer *to the line* in the update history in its SVB. Hence, the directory sets the P_r bit in its *Succ_Set* for P_s. Then, it sends to P_r the line from L2, the history, and P_s's ID.

In either case, P_r marks P_s as its predecessor by setting the correct bit in *Pred_Set* (as indicated in the message from the directory). Then, the history is inspected to find writers to the word that P_r is reading. The last one (say processor *j*), if any, is identified. In this case, P_r sets the *j* bit of its *Sq_Set* bitmask, since this is a RAW. In the future, if processor *j* is squashed, P_r will too.

The case of a read hit to a cached line is simple. There is no change to the cache coherence protocol states and, therefore, there is no change to the *Pred_Set* or *Succ_Set* of any processor. If the read hits in L0, no further action is taken because P_r reads its own update. Otherwise, the local SVB is checked for the line's update history. If it is found, it is checked for writers to the word that P_r is reading. The last one (say processor *j*), if any, is identified, and P_r sets the *j* bit of its *Sq_Set* bitmask, since this is a RAW. Note that this bit may already be set.

3.2.3. Write Access in a Transaction. We now describe the actions taken on a write (Figure 3(a)). Consider first a write that misses in the cache. The coherence protocol ensures that the non-speculative version of the line is brought to P_r's cache in M state, and all the sharers are invalidated. If there is a history of speculative updates, it is piggybacked on the same message and stored in P_r's SVB. In addition, all processors that had a copy of the line invalidate their relevant L0 and SVB entries.

There are two cases, depending on whether, before the write, the line was either in M state in the cache of the last writer (P_s), or in S state in multiple caches. In the first case, P_s combines the line's L0 and SVB entries into the update history, and sends to P_r the line, the history, and its ID. Also, P_s invalidates the line's entries in the cache, L0, and SVB, and sets the bit in its *Succ_Set* bitmask that corresponds to P_r, since P_r is its successor.

If, instead, the line was in S state in multiple caches, the directory module that maps the line (say *i*) processes the request. It sends invalidations to all sharers (including the last writer according to the SVB in the directory), passing P_r's ID. The sharer processors invalidate the line's entries in their cache, L0, and SVB, and set the bit in their *Succ_Set* bitmasks corresponding to P_r. Note that the last writer now has as successors directory *i* and the new writer. Finally, directory *i* clears its SVB entry but retains its *Succ_Set*. It responds to P_r with the line, the history, and list of the sharers in a bitmask.

As the response arrives at P_r, the *Pred_Set* is updated to include the old owner (in the first case) or the old sharers (in the second one). The update from P_r is kept in P_r's L0. Since P_r's access is a write, P_r's *Sq_Set* is unchanged.

Consider now a write hit to a line in S state. There is a coherence protocol transaction, where all the sharers get their entries for the line in their cache, L0, and SVB invalidated, and P_r transitions its line state to M. OmniOrder performs the same actions as in the case of a write miss on a line that was in S state remotely, except that, as the line and history arrive at P_r, they are discarded — P_r is already up-to-date.

Finally, on a write hit to a line in M state, OmniOrder performs no action. P_r already has the history (if it exists), the predecessor/successor state needs no change, and there is no need to set *Sq_Set* bits. The update is saved in L0, overwriting any update to the same word that was already there.

3.2.4. The Commit of a Transaction. A transaction in a processor (P) is ready to commit when both P has completed the execution of the transaction and P's *Pred_Set* is zero. The latter implies that all of P's predecessor transactions are either committed or squashed. Figure 3(b) outlines the commit algorithm. It has three operations: (i) merging all the speculative updates generated by P with the corresponding non-speculative lines, (ii) informing P's successor chain of the commit, starting with *Succ_Set* and *Succ_Dir*, which in turn may trigger the commit of other transactions, and (iii) clearing the local speculative hardware structures.

The first two operations happen concurrently. P's speculative updates are merged through a combination of local and remote actions. Specifically, consider all the updates generated by P while executing the transaction. Some of them are still in P's L0. These are the ones for lines that are in M or S state in P's cache. These updates are simply merged into P's cache. However, the lines in S state have copies in other caches (the co-sharer caches) and in L2. These copies also need to be updated. OmniOrder can access these co-sharers (and the L2) through the *Succ_Dir* bitmask, since they are P's successors. Consequently, as P sends commit signals to its successors in the second operation of the commit, the signals propagated through the *Succ_Dir* bitmask will reach the successor directories and, from those directories's *Succ_Set*, reach the co-sharer caches. In both directories and caches, OmniOrder will merge the corresponding SVB entries into the L2 and the caches, respectively. Finally, some of P's speculative updates are not in P's L0. They updated lines that have been invalidated from P's cache and, therefore, there is no local state. The updates were transferred (together with the line) to successor proces-

sors. Consequently, as P sends the commit signals through its successor chain, the signals propagated through P's *Succ_Set* will reach the processors with these updates in their SVBs. They will merge them into their caches.

From this discussion, we see that, as part of the commit, P sends a commit signal to all of its successors: the directory modules in its *Succ_Dir* and the processors in its *Succ_Set*. Both directories and processors further propagate the signal. On reception of the signal, a directory module inspects its SVB, picks the updates from P, merges them into the L2, clears these SVB updates, and propagates the signal to its *Succ_Set*. On reception of the signal, a processor checks its SVB, selects the updates from P, merges them into its cache, clears these SVB updates, and propagates P's commit signal through its own *Succ_Dir* and *Succ_Set*. After a processor completes its merge operations, if the signal was for the commit of one of the processors in its *Pred_Set*, then OmniOrder clears the corresponding *Pred_Set* bit. If, at this point, the whole *Pred_Set* is empty and the local processor has completed its own transaction, then the local processor starts its own commit.

Note that a processor may receive multiple commit signals corresponding to a single committing processor — e.g., due to successor links through two directory modules. The protocol handles this case gracefully without modification. Moreover, a processor may receive two commit signals from two processors out of order. This case is flagged when a processor inspects its SVB to process a commit signal. If it finds that the SVB contains an earlier update to the same line, the processor temporarily cancels this commit and waits for the missing commit signal, so it can process the commits in order.

Finally, the third operation in P's commit in Figure 3(b) is to clear the speculative hardware structures in P. They include *Succ_Set*, *Succ_Dir*, *Sq_Set*, and the bits in the cache tags that mark the lines in the cache that have been speculatively read or written. The L0 is already empty. The SVB is also empty because all the earlier transactions have already committed or been squashed. Similarly, *Pred_Set* is already cleared. Before the *Succ_Dir* gets cleared, however, P tells the directory modules in its *Succ_Dir* to clear their *Succ_Set* for P, since they will not be used anymore.

3.2.5. The Squash of a Transaction. A transaction (T) in a processor (P) is squashed when one of three scenarios happens. One is when P receives a coherence message that indicates that T may be participating in a dependence cycle with other transactions. This situation is discussed in Section 4.3. The second scenario is when an earlier transaction that is the source of a true dependence with T is squashed. Finally, T is squashed when it suffers conventional transaction-termination events such as exceptions or attempted cache overflow of lines with L0 or SVB state. We discuss the case of overflow in Section 4.2.

Figure 3(c) outlines the algorithm followed in the squash of a transaction in P. It comprises three operations: (i) discarding all the speculative updates generated by P, (ii) informing P's successor chain of P's squash, which may in turn trigger the squash of other transactions, and (iii) when all the earlier transactions are committed or squashed, clearing the local speculative hardware structures and restarting the transaction.

The first two operations happen concurrently. To discard the speculative updates generated by P, OmniOrder first clears all of P's L0 updates. Note that P's SVB never includes P's own updates. Therefore, the SVB should not be modified. In addition, no line in P's cache should be invalidated because the cache contains non-speculative data.

Some of P's speculative updates may have propagated to the SVBs of processors in P's successor chain. Consequently, P sends squash signals to the directory modules in its *Succ_Dir*, and to the processors in its *Succ_Set*. They, in turn, will propagate the signal to their successors.

When a directory module receives a squash signal originating in P, it inspects its SVB, discards the updates from P, and propagates the signal to its *Succ_Set* bitmask. Similarly, when a processor Q receives a squash signal originating in P, it checks its SVB, discards the updates from P, and propagates the signal further through its own *Succ_Set* and *Succ_Dir* bitmasks. After this, if Q's *Pred_Set* has the bit corresponding to P on, it is reset. In addition, if Q's *Sq_Set* has the bit corresponding to P on (i.e., P is the source of a true dependence with Q), then Q has to be squashed. Hence, Q initiates a squash for its own transaction.

Note that a processor Q may receive two squash signals from two earlier transactions out of order. This case requires no special support. Q can discard the SVB updates of the two earlier processors in any order — unlike for transaction commit, there is no need to do it in transaction order. In addition, Q may receive multiple squash signals corresponding to a single squashed processor. The protocol handles this case with no changes as in the commit case.

Finally, the third operation in P's squash as shown in Figure 3(c) only takes place when P's *Pred_Set* is found to be empty — which means that all of P's predecessor transactions have committed or been squashed. At this point, OmniOrder clears the speculative hardware structures in P. They include *Succ_Set*, *Succ_Dir*, *Sq_Set*, and the speculative access bits in the cache tags. The L0 is already cleared. The SVB is already empty because there is no earlier transaction than P. Like in the commit, before the *Succ_Dir* gets cleared, P requests the directory modules in its *Succ_Dir* bitmask to clear their *Succ_Set* for P. At this point, P can restart the transaction.

4. OmniOrder Implementation

This section describes a possible implementation of OmniOrder. We first describe OmniOrder's hardware structures, and then how OmniOrder handles speculative state overflow and detects conflict cycles.

4.1. Hardware Structures

4.1.1. Hardware Structures in the Processor. As shown in Figure 4(a), OmniOrder's main hardware structures in the processor are the L0, the SVB, and a set of bitmasks in the cache controller. The L0 is a small and fast set-associative

cache that contains the speculative updates generated by the local processor — all except for updates to lines that have been invalidated from the L1 cache. L0 lines have the same size as L1 cache lines, but they add per-word valid bits (or per-byte bits, instead, if the finest granularity of access used is bytes). The L0 is accessed before the L1 cache and ensures that locally-produced speculative data is readily accessible.

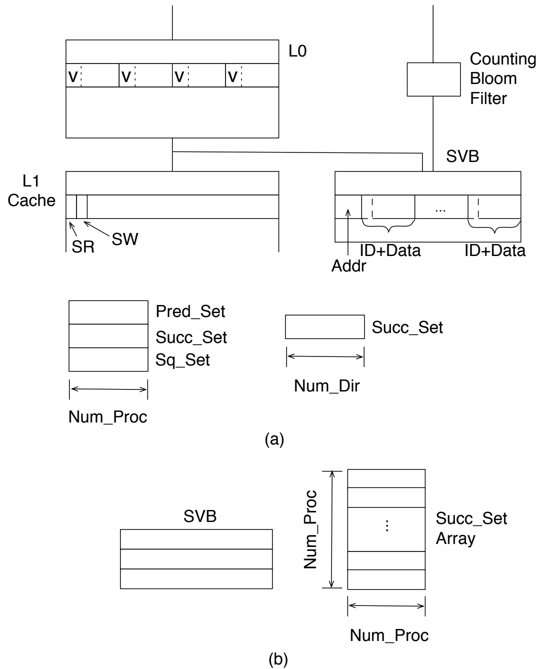

Figure 4: OmniOrder hardware structures.

The SVB is a small fully-associative buffer that contains the history of speculative updates to locally-cached lines by other processors executing transactions that are ordered before. Each entry corresponds to one word-level address. As shown in the figure, an entry contains the word address and the history of updates to it. The history consists of up to four *ordered* updates from different processors. Each update has the writing processor ID and the last value produced by that processor. The SVB is accessed at the same time as the L1 cache.

Since it is likely that there are only a few dependences between concurrently-executing transactions, the SVB is small. However, it would still be costly to access the SVB every time that the L1 is accessed. Hence, to save SVB accesses, we add a counting bloom filter [9]. The filter has the hashes of all the addresses currently in the SVB. Since it is a counting filter, it allows adding and removing addresses as they are added and removed from the SVB. The filter is accessed in parallel with L0; if it misses, the SVB will not be accessed.

After an SVB entry receives its initial contents, its updates will be gradually removed as transactions commit or get squashed. No new updates will ever be added, until the entry is completely cleared and reloaded. Specifically, updates by committing transactions will be removed in order; updates by squashed transactions will be removed in any order, potentially creating "holes" in the history. Overall, given an SVB entry, it is always easy to deduce the order of the updates. Finally, given the likely modest number of dependences between concurrently-executing transactions, each SVB entry only needs space for a few updates (e.g., four).

The cache controller has four bitmasks. *Pred_Set*, *Succ_Set*, and *Sq_Set* have as many bits as processors in the manycore. They record the processors whose transactions immediately precede (*Pred_Set*) or succeed (*Succ_Set*) the local one, and the processors whose transactions, if squashed, cause the current one to squash (*Sq_Set*). *Succ_Dir* has as many bits as directory modules in the manycore. It records the directory modules that act as successors of the current transaction.

Finally, the tag of each L1 cache line has a Speculative Read (SR) and a Speculative Written (SW) bit. These bits are set when the line is read or written, respectively, by the local transaction. When an incoming coherence transaction is received for a line, these bits are checked to decide whether the L0 and SVB need to be accessed.

4.1.2. Hardware Structures in the Directory Module. As shown in Figure 4(b), the hardware structures in each directory module are an SVB and a *Succ_Set* array. The SVB is like the one in a processor. The *Succ_Set* array contains one *Succ_Set* for each of the processors in the manycore. Each *Succ_Set* is like the one in a processor.

4.1.3. Summary. OmniOrder's hardware needs are modest. In addition, the hardware is relatively decoupled from the cache coherence protocol. Cache lines transition between the conventional coherence states. Also, requests issue line-level addresses to the network, and move full cache lines rather than single words. All this keeps OmniOrder's complexity modest. We will also see that the increase in traffic is very small.

4.2. Handling Speculative State Overflow

If an line in the L1 cache that has been accessed in a transaction is about to be displaced from the hardware structures due to overflow, the transaction cannot continue normally. However, we have to be careful about naively squashing the transaction: the potentially displaced line may have a history of earlier updates in its SVB, and such updates cannot be discarded or lost. For this reason, in OmniOrder, an overflow can only discard a line accessed in the transaction if the line has no local SVB entries, and it immediately triggers a squash. A special case occurs when all the lines in a cache set have SVB state. In this case, an access to another line mapping to the same set immediately triggers a squash.

After the overflow-triggered squash, the transaction is restarted in a *conventional* transactional mode that does not allow inter-transactional dependences. This approach is similar to that of Ramadan et al. [20]. Once in conventional transactional mode, on speculative state overflow, a transaction has not exposed its state to other transactions or used other transactions' state. Hence, OmniOrder can avoid squashing it by using any of the existing conventional overflow-handling TM mechanisms (e.g., [11]).

4.3. Conflict Cycle Detection

When conflicting transactions form a cycle, they cannot be serialized any more. As an example, consider Figure 5, which shows four transactions running concurrently. Conflicts between transactions are shown as arrows. If, at this point a conflict between T2 a T3 appears, there is a cycle. Consequently, OmniOrder has to detect each cycle, break the cycle by squashing at least one of the conflicting transactions, and ensure that the cycle cannot repeatedly reappear.

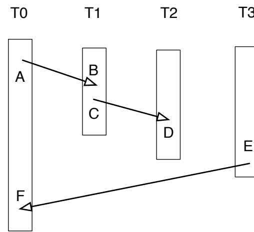

Figure 5: Example of transactions with conflicts.

OmniOrder uses a cheap, conservative way to detect and break cycles. It uses the fact that, to participate in a cycle, a transaction has to be the source of a line dependence and the destination of another line dependence. In addition, in at least one of the transactions in the cycle, the dependence where the transaction is the source is formed before the dependence where the transactions is the destination. For example, in Figure 5, if a new dependence from from T2 to T3 is created, the condition is true for at least T0. However, this criterion is conservative, as the condition may be true without a cycle — e.g., in Figure 5, the condition is true for T0 but there is no cycle.

Hence, when a transaction (T) becomes the destination of a conflict, OmniOrder checks T's *Succ_Set* and *Succ_Dir*. If they are not empty, then there is a chance of a cycle and OmniOrder triggers the squash of T (Section 3.2.5). If there is a cycle, the squash signal eventually returns to T, which ignores it.

We want to avoid that, as squashed transactions restart, they fall into the same cycle again. Hence, the single transaction (T) that detected the cycle is restarted in the conventional transactional mode that does not allow inter-transactional dependences (Section 4.2). Consequently, T will not participate in the cycle again, and the same cycle cannot reoccur.

Overall, compared the cycle-detection algorithm described in Volition [19], OmniOrder uses trivially-simple hardware and minimizes protocol races because only a single processor detects the cycle. However, while Volition finds only true cycles, OmniOrder can report false-positive cycles.

5. Speculation for Strict Consistency Models

OmniOrder also allows the concurrent execution of conflicting hardware-generated atomic blocks. The protocol needs no changes. Here, we describe how OmniOrder eliminates the stalls of loads and stores under SC. The result is a more powerful environment than existing proposals for out-of-window speculation (e.g., InvisiFence [6] and Bulk [24]).

5.1. Basic Idea

In a basic implementation of SC, a store in the write buffer prevents subsequent loads from retiring and subsequents writes in the write buffer from performing. As a result, the processor may stall. Current proposals for out-of-window speculation avoid stalling by creating a register checkpoint in hardware before the disallowed access, and continuing execution speculatively, typically buffering the speculative state in the cache. For example, InvisiFence triggers a checkpoint and enters speculative execution when a load or a store are about to retire and there is already a store in the write buffer. Eventually, when the write buffer becomes empty, the speculative block of instructions executed is committed, the checkpoint is discarded, and the execution returns to non-speculative mode. However, if before this happens, the processor (P0) receives a coherence message on any speculative data, the speculative block is squashed and the register checkpoint is restored. As an alternative to the squash, in InvisiFence with Commit on Violation (CoV), P0 delays processing the coherence action and stalls, hoping to drain the write buffer and commit all speculative data before processing the incoming signal. If the speculative block has not committed after some timeout, it is squashed.

OmniOrder goes further (Figure 6). If, while P0 is executing block B0 speculatively, it receives a coherence message from another processor (P1) on any speculative data, P0 is not squashed. Instead, P1 starts its own speculative block (B1) that is made a successor of B0. These two speculative blocks follow the same OmniOrder protocol as in Section 3, and will be forced to commit in order. Other speculative blocks from other processors can also join in, creating successor chains. Compared to InvisiFence, OmniOrder has avoided the squash, hence improving performance.

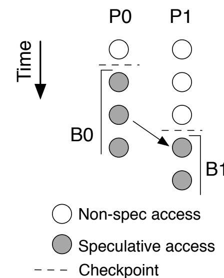

Figure 6: Hardware-initiated speculative blocks.

5.2. Starting and Terminating Speculative Execution

To support the concurrent execution of conflicting speculative blocks, OmniOrder extends the conditions required to start and terminate speculative execution beyond InvisiFence's. Specifically, in OmniOrder, processors enter *Speculative Mode* for one of two reasons. The first one is to avoid stalling at loads or stores when SC would require it — exactly like in InvisiFence. The second reason is the execution of an access that is found to be data dependent on a reference in another processor's speculative block. To understand the implications of this second reason, we consider loads and stores separately.

Assume that processor P1 is not executing a speculative block. Assume that it executes a load and that the return line brings the history of speculative updates (just one in this case)

from P0, which is executing B0. In this case, OmniOrder starts a dependent speculative block B1 in P1. For this, it marks the ROB entry for the load so that, right before the load retires, it creates a register checkpoint. The load is the first instruction of speculative block B1, which is a successor of speculative block B0 in P0 (Figure 6).

Assume now that, instead, P1 executes a store and the response message brings the history of speculative updates from P0. Unlike in the previous case, the store has already retired. Now, we would not be able to create a register checkpoint, which is needed in order to support speculative blocks. Consequently, OmniOrder *does* create a checkpoint every time that a store retires and the processor is not already in Speculative Mode. With this design, the store is the first instruction of speculative block B1, which is a successor of B0 (Figure 6).

In OmniOrder, for a processor to commit a block B1 and leave the Speculative Mode, two conditions must simultaneously hold. The first one is that the local write buffer is empty. This is the one and only condition needed in InvisiFence. The second condition is that all its predecessor speculative blocks (such as B0 in Figure 6) have committed or been squashed. The processor can help these conditions by stalling when B1 is getting large and, hence, there is risk of losing much work.

Overall, the complete OmniOrder rules for checkpointing and entering Speculative Mode is as follows. For a store, checkpoint at retirement. In this case, the write buffer is necessarily empty since, otherwise, the processor would already be in Speculative Mode. This rule supersedes all the checkpoints by InvisiFence at writes. For a load, checkpoint at retirement if either the response included a history of speculative updates, or the write buffer is not empty. This second requirement is needed to avoid SC-induced stall, and is also present in InvisiFence.

Compared to a scheme like InvisiFence, OmniOrder activates the Speculative Mode more often. While some of these additional activations are needed (e.g., when the processor accesses data that is speculative elsewhere), others are not. Unnecessary activations occur when a store (i) finds an empty write buffer, (ii) does not end up accessing a speculatively line elsewhere, and (iii) completes and gets removed from the write buffer before the processor retires any subsequent load or store (otherwise, a checkpoint is needed to avoid an SC-induced stall). In practice, these stores cause little overhead, since, as soon as they complete, execution leaves Speculative Mode.

5.3. Speculative State Overflow

Handling speculative state overflow in this environment is relatively simple: before any speculative state is displaced, the processor stalls. If this was a speculative block generated to avoid SC-induced stalls, it will eventually commit as its pending stores complete. If, instead, this block was started due to a dependence, once its predecessors commit or get squashed, and its pending stores complete, it will be able to commit. This is unlike when executing transactions (Section 4.2), where atomic blocks cannot commit early.

6. Evaluation

We evaluate OmniOrder using simulations of a 64-processor chip using the SESC simulator [22]. The architecture modeled is shown in Table 1. Note that, while Section 3 described OmniOrder's operation under an MSI protocol, we model an MESI protocol. We focus on three aspects: (1) the performance of OmniOrder with TM, (2) the performance of OmniOrder with speculation for SC, and (3) the overhead of OmniOrder.

Parameter	Value
Architecture	Manycore chip with 64 cores
Core	4-issue wide out-of-order
ROB, write buffer	176 entries, 32 entries
Memory consistency	RC or TSO at the hardware level
L0 buffer	8KB, 4-way asso.
SVB	128 entries/proc; 128 entries/dir module
Priv. L1 cache	64KB WB, 4-way asso., 2-cycle round trip
Shar. L2 cache	256KB per bank, 64 banks, 8-way asso., 11-cycle round trip (local module)
Cache line size	32B
Coherence	MESI, full-mapped directory in 64 modules
Network	2-D mesh, 7-cycle hop latency
Main memory	200-cycle round trip

Table 1: Architecture simulated.

For (1), we use 8 applications from STAMP. We compare OmniOrder to InvisiFence, which squashes a transaction on conflict. For InvisiFence, we use the "oldest transaction wins" policy. For (2), we compare OmniOrder to 5 existing SC implementations on top of RC hardware. The schemes are shown in Table 2. For END_SC [23], we model a system with 16KB pages. For each scheme, we use 12 applications from SPLASH-2 and 4 from PARSEC. To get further insight, we also use the three most competitive schemes (IF, IF_COV and OO), to run 11 small programs that implement concurrent algorithms such as non-blocking queues. These codes were mostly obtained from [1]. Finally, for (3), we measure the communication overhead of OmniOrder. Table 3 shows the applications.

Name	Implementation Description
SC_Window	In-window speculation [12]
CO	Conflict Ordering [17]
END_SC	End-To-End SC [23]
IF	InvisiFence [6]
IF_COV	IF with Commit-On-Violation [6]
OO	OmniOrder

Table 2: SC implementations on top of RC hardware.

6.1. Performance of OmniOrder with TM

Figure 8 compares the execution time of the STAMP applications running on InvisiFence (IF) and OmniOrder (OO). For each program, the execution time is normalized to IF, and the cycles broken into the categories: retiring instructions (*Useful*), stalled due to pipeline hazards (*Pipeline*), stalled due to memory system accesses (*Memory*), stalled due to store buffer full (*SBFull*), and squashed (*Squashed*). The rightmost two bars show the averages, and are not broken into categories.

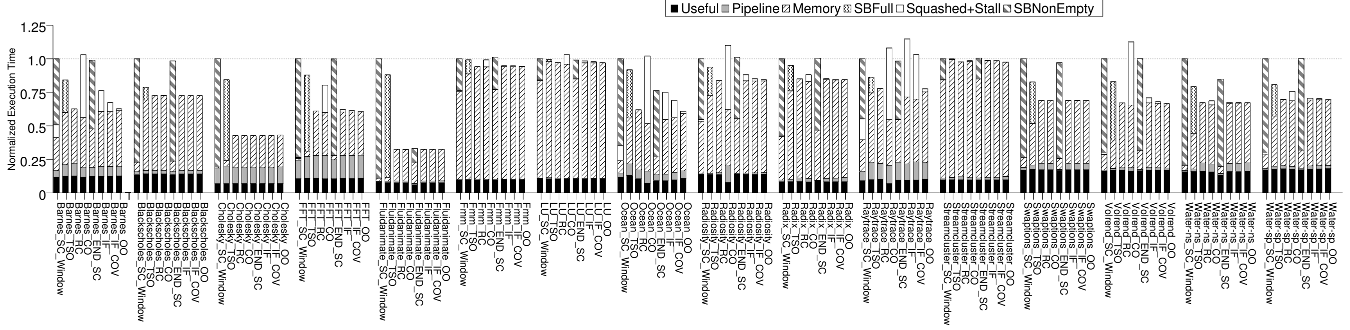

Figure 7: Execution time for different implementations of SC, TSO, and RC.

Set	Application	Description
Conc. Algo.	Aharr	Variant of Harris
	Dekker	Algorithm for 2 proc. mutual exclusion
	Harris	Non-blocking set
	Lazylist	List-based concurrent set
	Moirbt	Non-blocking sync. primitives
	Moircas	Non-blocking sync. primitives
	Ms2	Two-lock queue
	Msn	Non-blocking queue
	Mst	Non-blocking queue
	Peterson	Algorithm for N proc. mutual exclusion
	Snark	Non-blocking double-ended queue
Apps	12 SPLASH-2, 4 PARSEC, and 8 STAMP codes	

Table 3: Applications executed.

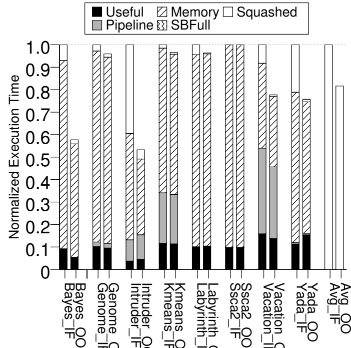

Figure 8: Execution time of OmniOrder with TM.

We see that, on average, OO reduces the execution time of the applications by a significant 18.4% over IF. These applications have little Useful time; most of the time is spent on Memory, Pipeline, and Squashed. OmniOrder is able to speed-up some programs mostly by reducing the Squashed time through transaction serialization. This is especially significant in Intruder, which has a producer-consumer pattern between some transactions. In other programs, OmniOrder speeds-up execution mostly by reducing the Memory time. This is done by allowing speculative data to be used by dependent processors, rather than squashing and accessing the data from memory. These results are similar to [20].

6.2. Performance of OmniOrder for Efficient SC

Figure 7 compares the execution time of the SPLASH-2 and PARSEC applications running on 6 implementations of SC on top of RC hardware (including OmniOrder (OO)). For each application, the execution time is normalized to SC_Window. As a reference, the figure also shows the time for TSO and RC executions. The execution cycles are broken into the same categories as in Figure 8 plus a new category: stalled due to non-empty store buffer (*SBNonEmpty*). To simplify the plot, in IF_COV, the stall time is included in the Squashed cycles.

The leftmost three bars show that the execution time decreases from SC_Window to TSO and to RC. The next two schemes (CO and END_SC) perform optimized in-window speculation. CO reduces the execution time by lowering the wait time for store completion. However, some eagerly-fetched pending lists force some load hits to re-execute after invalidating cache lines. END_SC often suffers from SBNonEmpty cycles. This is because many accesses use the FIFO store buffer, since they are classified as unsafe.

The last three schemes (IF, IF_COV, and OO) perform out-of-window speculation. For most programs, their execution time is similar, and very close to RC. This is because there are few block conflicts. Still, in Barnes, Ocean, and Raytrace, there are conflicts, and IF incurs a visible amount of Squashed cycles. IF_COV reduces the Squashed+Stall cycles by a certain amount, and OO practically removes them all. On average, however, the three schemes perform similarly here.

Figure 9 shows the execution time of the concurrent algorithms for the best three schemes (IF, IF_COV and OO). These programs have more frequent block conflicts. Hence, IF has significant Squashed time. IF_COV avoids some squashes by stalling, but the combination Squashed+Stall is still high. In fact, stalling a block may cause more blocks to get trapped in the stall. OO eliminates practically all the Squashed time, reducing the execution time noticeably. On average, OO's execution time is 15.3% lower than IF_COV's.

Overall, we conclude that, for codes with very few conflicts, IF and OO perform similarly. However, when there are frequent conflicts, as in three of our applications and all of our concurrent algorithms, OO provides a good speedup over IF and IF_COV. The data shows that most of these conflicts do not create cycles, and that both squashes and stalls can be avoided.

6.3. OmniOrder Communication Overhead

To assess OmniOrder's communication overhead, Figure 10 shows all the traffic in the system in bytes broken down into the

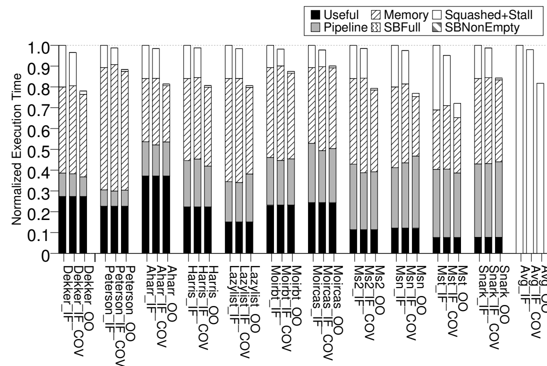

Figure 9: Execution time of concurrent algorithms.

following categories: *MemAcc* is the off-chip traffic; *Read* and *Write* are the on-chip traffic related to read and write requests, respectively; and *Fwd* is the traffic due to forwarding the update histories and squash/commit signals between processors. Each bar is normalized to the sum of MemAcc, Read, and Write. In all programs, the additional bandwith consumed by OmniOrder with Fwd is very small.

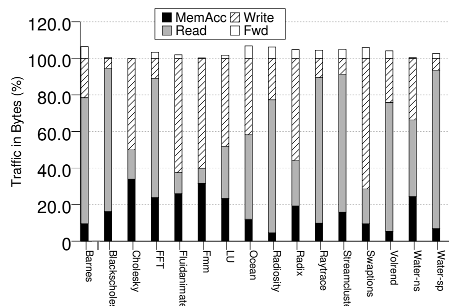

Figure 10: Traffic in the system.

7. Related Work

OmniOrder builds on extensive work on TM [14]. On a conflict, most TM schemes squash one of the transactions, while a few stall one of the transactions. RETCON [7] allows conflicting transactions to complete, and then employs slice re-execution to repair incorrect memory state. Such repair is limited to simple cases such as counters. Other proposals learn repeated conflicts and avoid scheduling conflicting transactions in parallel [5] or take checkpoints to reduce the rollback extent [25].

Enforcing strict memory consistency models through out-of-window speculation has received much attention (e.g., [6, 10, 13, 21, 26]). This environment is different from TM, since atomic blocks are created dynamically by the hardware, and can typically terminate as soon as all the pending accesses commit. OmniOrder is the first scheme that allows these atomic blocks to provide speculative data to other processors. Other schemes retain SC without needing out-of-window speculation — e.g., Conflict Ordering [17] tries to avoid dependence cycles that could violate SC, while End-to-End SC [23] directs accesses to private and shared (or unsafe) data to different write buffers, and only reorders the former.

There is a similarity between the history of updates transferred in OmniOrder and the *diff* modifications transferred in software DSM systems that use lazy RC such as TreadMarks [3]. A *diff* encodes the modifications to a page.

8. Conclusion

This paper presented OmniOrder, the first design that efficiently executes conflicting atomic blocks concurrently in a directory-based coherence environment. Caches keep only non-speculative data, and coherence protocol transfers include the history of speculative updates to the line. We evaluated OmniOrder with 64-core simulations. In a TM environment, OmniOrder reduced the execution time of the STAMP programs by an average of 18.4% over a scheme that squashed on conflict. In an environment for SC enforcement, on average, SPLASH-2 and PARSEC programs have too few conflicting blocks for OmniOrder to make a difference. However, we ran 11 concurrent-algorithm programs that and have frequent speculation. OmniOrder reduced their execution time by an average of 15.3% over a scheme that squashed on conflict. Finally, the communication overhead of OmniOrder was negligible.

References

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

[2] W. Ahn et al. BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO'09*.

[3] C. Amza et al. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, February 1996.

[4] U. Aydonat and T. Abdelrahman. Hardware Support for Relaxed Concurrency Control in Transactional Memory. In *MICRO*, 2010.

[5] G. Blake et al. Proactive Transaction Scheduling for Contention Management. In *MICRO*, December 2009.

[6] C. Blundell et al. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *ISCA*, 2009.

[7] C. Blundell et al. RETCON: Transactional Repair without Replay. In *ISCA*, June 2010.

[8] J. Bobba et al. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, June 2008.

[9] F. Bonomi et al. An Improved Construction for Counting Bloom Filters. In *European Symposium on Algorithms*, Sep 2006.

[10] L. Ceze et al. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, June 2007.

[11] J. Chung et al. Tradeoffs in Transactional Memory Virtualization. In *ASPLOS*, 2006.

[12] K. Gharachorloo et al. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP*, August 1991.

[13] C. Gniady et al. Is SC + ILP = RC? In *ISCA*, May 1999.

[14] T. Harris et al. *Transactional Memory*. Morgan & Claypool, 2010.

[15] M. Herlihy and E. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA*, 1993.

[16] S. A. R. Jafri et al. Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies. In *ASPLOS*, 2013.

[17] C. Lin et al. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, 2012.

[18] X. Qian et al. BulkSMT: Designing SMT Processors for Atomic-Block Execution. In *HPCA*, February 2012.

[19] X. Qian et al. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *ASPLOS*, 2013.

[20] H. Ramadan et al. Dependence-Aware Transactional Memory for Increased Concurrency. In *MICRO*, 2008.

[21] P. Ranganathan et al. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *SPAA*, June 1997.

[22] J. Renau et al. SESC Simulator. http://sesc.sourceforge.net.

[23] A. Singh et al. End-To-End Sequential Consistency. In *ISCA*, 2012.

[24] J. Torrellas et al. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, 52(12), 2009.

[25] M. Waliullah et al. Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems. In *IPDPS*, 2008.

[26] T. Wenisch et al. Mechanisms for Store-wait-free Multiprocessors. In *ISCA*, June 2007.