BulkSC: Bulk Enforcement of Sequential Consistency*

Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas

Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu
{luisceze,jtuck,pmontesi,torrellas}@cs.uiuc.edu

Abstract

While Sequential Consistency (SC) is the most intuitive memory consistency model and the one most programmers likely assume, current multiprocessors do not support it. Instead, they support more relaxed models that deliver high performance. SC implementations are considered either too slow or — when they can match the performance of relaxed models — too difficult to implement.

In this paper, we propose *Bulk Enforcement of SC (BulkSC)*, a novel way of providing SC that is simple to implement and offers performance comparable to Release Consistency (RC). The idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. The hardware enforces SC at the coarse grain of chunks which, to the program, appears as providing SC at the individual memory access level. BulkSC keeps the implementation simple by largely *decoupling* memory consistency enforcement from processor structures. Moreover, it delivers high performance by enabling full memory access reordering and overlapping within chunks and across chunks. We describe a complete system architecture that supports BulkSC and show that it delivers performance comparable to RC.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—*MIMD processors*; D.1.3 [Programming Techniques]: Concurrent Programming—*parallel programming*

General Terms: Design, Performance

Keywords: Memory Consistency Models, Chip Multiprocessors, Programmability, Bulk, Sequential Consistency

1. Introduction

With chip multiprocessors now becoming ubiquitous, there is growing expectation that parallel programming will become popular. Unfortunately, the vast majority of current application programmers find parallel programming too difficult. In particular, one of the trickiest aspects of parallel programming is understanding the effects of the memory consistency model supported by the machine on program behavior. The memory consistency model specifies what orderings of loads and stores may occur when several processes are accessing a common set of shared-memory locations [1].

It is widely accepted that the most intuitive memory consistency model, and the one that most programmers assume is Sequential Consistency (SC) [21]. SC requires that all memory operations of all processes appear to execute one at a time, and that the operations of a single process appear to execute in the order described by that process's program. Programmers prefer this model because it offers the same relatively simple memory interface as a multitasking uniprocessor [18].

Despite this advantage of SC, manufacturers such as Intel, IBM, AMD, Sun and others have chosen to support more relaxed memory consistency models [1]. Such models have been largely defined and used to facilitate implementation optimizations that enable memory access buffering, overlapping, and reordering. It is felt that a straightforward implementation of the stricter requirements imposed by SC on the outstanding accesses of a processor impairs performance too much. Moreover, it is believed that the hardware extensions that are required for a processor to provide the illusion of SC at a performance competitive with relaxed models are too expensive.

To ensure that the upcoming multiprocessor hardware is attractive to a broad community of programmers, it is urgent to find novel implementations of SC that both are simple to realize and deliver performance comparable to relaxed models. In this paper, we propose one such novel implementation.

We call our proposal *Bulk Enforcement of SC* or *BulkSC*. The key idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. Then, the hardware enforces SC at the coarse grain of chunks rather than at the conventional, fine grain of individual memory accesses. Enforcing SC at chunk granularity can be realized with simple hardware and delivers high performance. Moreover, to the program, it appears as providing SC at the memory access level.

BulkSC keeps hardware simple mainly by leveraging two sets of mechanisms: those of Bulk operations [8] and those of checkpointed processors (e.g., [3, 7, 10, 19, 23, 24, 29]). Together, they largely *decouple* memory consistency enforcement from processor structures. *BulkSC* delivers high performance by allowing full memory access reordering and overlapping within chunks and across chunks.

In addition to presenting the idea and main implementation aspects of *BulkSC*, we describe a complete system architecture that supports it with a distributed directory and a generic network. Our results show that *BulkSC* delivers performance comparable to Release Consistency (RC) [13]. Moreover, it only increases the network bandwidth requirements by 5-13% on average over RC, mostly due to signature transfers and squashes.

This paper is organized as follows. Section 2 gives a background; Section 3 presents Bulk Enforcement of SC; Sections 4, 5, and 6 describe the complete architecture; Section 7 evaluates it; and Section 8 discusses related work.

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

2. Background

2.1. Sequential Consistency

As defined by Lamport [21], a multiprocessor supports SC if the result of any execution is the same as if the memory operations of all the processors were executed in some sequential order, and those of each individual processor appear in this sequence in the order specified by its program. This definition comprises two ordering requirements:

Req1. Per-processor program order: the memory operations from individual processors maintain program order.

Req2. Single sequential order: the memory operations from all processors maintain a single sequential order.

SC provides the simple view of the system shown in Figure 1. Each processor issues memory operations in program order and the switch connects an arbitrary processor to the memory at every step, providing the single sequential order.

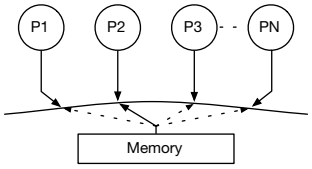


Figure 1. Programmer's model of SC.

A straightforward implementation of SC involves satisfying the following requirements [1]: (i) a processor must ensure that its previous memory operation is complete before proceeding with its next one in program order, (ii) writes to the same location need to be made visible in the same order to all processors, and (iii) the value of a write cannot be returned by a read until the write becomes visible to all processors — for example, when all invalidations or updates for the write are acknowledged. Since requirement (i), in particular, limits performance significantly, several optimizations have been proposed to enable memory accesses to overlap and reorder while keeping the illusion of satisfying these requirements.

Gharachorloo *et al*. [12] proposed two techniques. The first one is to automatically prefetch ownership for writes that are delayed due to requirement (i). This improves performance because when the write can be issued, it will find the data in the cache — unless the location is invalidated by another thread in between. The second technique is to speculatively issue reads that are delayed due to requirement (i) — and roll back and reissue the read (and subsequent operations) if the line read gets invalidated before the read could have been originally issued. This technique requires an associative load buffer that stores the speculatively-read addresses. Incoming coherence requests and local cache displacements must snoop this buffer, and flag an SC violation if their address matches one in the buffer. The MIPS R10000 processor supported SC and included this technique [31]. Later, Cain *et al*. [6] proposed an alternate implementation of this technique based on re-executing loads in program order prior to retirement.

Ranganathan *et al*. [27] proposed Speculative Retirement, where loads and subsequent memory operations are allowed to speculatively retire while there is an outstanding store, although requirement (i) would force them to stall. The scheme needs a history buffer that stores information about the speculatively retired instructions. The per-entry information includes the access address, the PC, a register, and a register mapping. Stores are not allowed to get reordered with respect to each other. As in the previous scheme, the buffer is snooped on incoming coherence actions and cache displacements, and a hit is a consistency violation that triggers an undo.

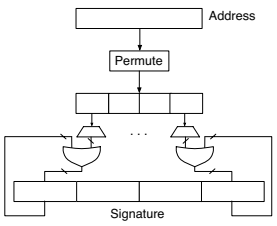
Gniady *et al*. [15] proposed *SC++*, where both loads and stores can be overlapped and reordered. The ROB is extended with a similar history buffer called Speculative History Queue (SHiQ). It maintains the speculative state of outstanding accesses that, according to requirement (i), should not have been issued. To reduce the cost of checking at incoming coherence actions and cache displacements, the scheme is enhanced with an associative table containing the different lines accessed by speculative loads and stores in the SHiQ. Since the SHiQ can be very large to tolerate long latencies, in [14], they propose *SC++lite*, a version of SC++ that places the SHiQ in the memory hierarchy.

While these schemes progressively improve performance — SC++ is nearly as fast as RC — they also increase hardware complexity substantially because they require (i) associative lookups of sizable structures and/or (ii) tight coupling with key processor structures such as the load-store queue, ROB, register files, and map tables.

2.2. Bulk

Bulk [8] is a set of hardware mechanisms that simplify the support of common operations in an environment with multiple speculative tasks such as Transactional Memory (TM) and Thread-Level Speculation (TLS). A hardware module called the Bulk Disambiguation Module (BDM) dynamically summarizes into Read (R) and Write (W) signatures the addresses that a task reads and writes, respectively. Signatures are $\approx$2 Kbit long and are generated by accumulating addresses using a Bloom filter-based [4] hashing function (Figure 2(a)). Therefore, they are a superset encoding of the addresses. When they are communicated, they are compressed to $\approx$350 bits.

The BDM also includes units that perform the basic signature operations of Figure 2(b) in hardware. For example, intersection and union of two signatures perform bit-wise AND and OR operations. The combination of the decoding (δ) and membership ($\in$) operations provides the *signature expansion* operation. This operation finds the set of lines in a cache that belong to a signature without traversing the cache. It is used to perform *bulk invalidation* of the relevant lines from a cache.



(a)

Op.	Description
$\cap$	Signature intersection
$\cup$	Signature union
$= \emptyset$	Is signature empty?
$\in$	Membership of an address in signature
δ	Signature decoding into cache sets

(b)

Figure 2. Signature encoding (a) and primitive operations (b).

Bulk has been proposed for environments with speculative tasks such as TM or TLS that perform conflict detection between tasks only when a task tries to commit [8]. Speculatively-written lines are kept in-place in the cache and cannot be written back before commit. Speculatively-read lines can be displaced at any time because the R signature keeps a record of the lines read. We use this same environment in this paper.

In this environment, signature operations are simple building blocks for several higher-level operations. As an example, consider the commit of task *C*. The BDM in the task's processor sends its W_C to other processors. In each processor, the BDM performs *bulk disambiguation* to determine if its local task *L* collides with *C*, as follows: $(W_C \cap R_L) \cup (W_C \cap W_L)$. If this expression does not resolve to empty, *L* should be squashed. In this case, the BDM performs bulk invalidation of the local cache using W_L, to invalidate all lines speculatively written by *L*. Finally, irrespective of whether *L* is squashed, the BDM performs bulk invalidation of the local cache using W_C, to invalidate all lines made stale by *C*'s commit. In the whole process, the BDM communicates with the cache controller, and the cache is completely unaware of whether it contains speculative data; its data and tag arrays are unmodified.

3. Bulk Enforcement of Sequential Consistency

Our goal is to support the concept of SC expressed in Figure 1 with an implementation that is simple and enables high performance. We claim that this is easier if, rather than conceptually turning the switch in the figure at individual memory access boundaries, we do it only at the boundaries of groups of accesses called Chunks. Next, we define an environment with chunks and outline a chunk-based implementation of SC.

3.1. An Environment with Chunks

Consider an environment where processors execute sets of consecutive dynamic instructions (called *Chunks*) as a unit, in a way that each chunk appears to execute atomically and in isolation. To ensure this perfect encapsulation, we enforce two rules:

Rule1. Updates from a chunk are not visible to other chunks until the chunk completes and commits.

Rule2. Loads from a chunk have to return the same value as if the chunk was executed at its commit point. Otherwise, the chunk would have "observed" a changing global memory state while executing.

In this environment, where a chunk appears to the system and is affected by the system as a single memory access, we will support SC if the following "chunk requirements" — which are taken from Section 2.1 using chunks instead of memory accesses — hold:

CReq1. Per-processor program order: Chunks from individual processors maintain program order.

CReq2. Single sequential order: Chunks from all processors maintain a single sequential order.

In this environment, some global interleavings of memory accesses from different processors are not possible (Figure 3). However, all the resulting possible executions are sequentially consistent at the individual memory access level.

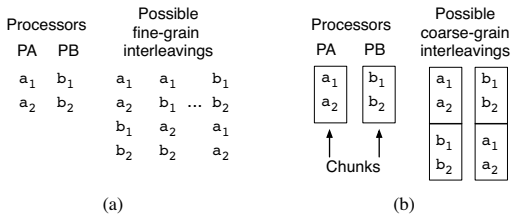


Figure 3. Fine (a) and coarse-grain (b) access interleaving.

3.2. Implementing SC with Chunks

A trivial implementation that satisfies these two SC requirements involves having an arbiter module in the machine that commands processors to execute for short periods, but only one processor at a time. During the period a processor runs, it executes a chunk with full instruction reordering and overlapping. In reality, of course, we want all processors to execute chunks concurrently. Next, we describe a possible implementation, starting with a naive design and then improving it.

3.2.1. Naive Design

We divide our design into two parts, namely the aspects necessary to enforce the two rules of Section 3.1, and those necessary to enforce the two chunk requirements of SC from Section 3.1.

Enforcing the Rules for Chunk Execution. To enforce *Rule1*, we use a cache hierarchy where a processor executes a chunk speculatively as in TLS or TM, buffering all its updates in its cache. These buffered speculative updates can neither be seen by loads from other chunks nor be displaced to memory. When the chunk commits, these speculative updates are made visible to the global memory system.

To enforce *Rule2*, when a chunk that is executing is notified that a location that it has accessed has been modified by a committing chunk, we squash the first chunk.

To see how violating *Rule2* breaks the chunk abstraction, consider Figure 4(a). In the figure, *C1* in processor *P1* includes *LD A* followed by *LD B*, while *C2* in *P2* has *ST A* followed by *ST B*. Suppose *P1* reorders the loads, loading *B* before *C2* commits. By *Rule1*, *C1* gets committed data, namely the value before *ST B*. If *C2* then commits — and therefore makes its stores visible — and later *C1* loads *A*, then *C1* would read the (committed) value of *A* generated by *C2*. We would have broken the chunk abstraction.

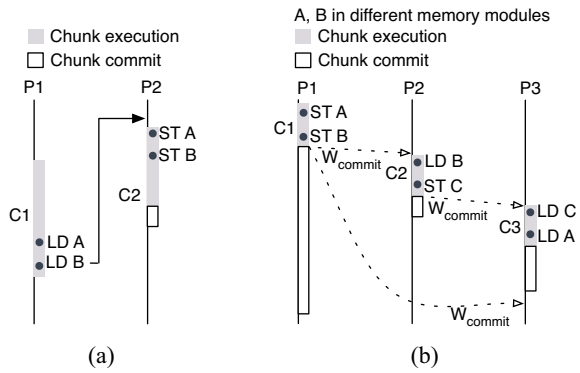


Figure 4. Examples of access reordering.

We efficiently enforce *Rule2* with Bulk Disambiguation [8]. When a chunk commits, it broadcasts its write signature (W_{commit}), which includes the addresses that it has updated. Any chunk *C1* that is executing intersects W_{commit} with its own *R* and *W* signatures[1]. If the intersection is not empty (there is a *collision*), *C1* is squashed and re-executed.

Enforcing the Chunk Requirements of SC. Given two consecutive chunks C_{pred} and C_{succ} from an individual processor, to enforce *CReq1* (per-processor program order requirement), we need

[1] Given that a store updates only part of a cache line, W_{commit} is intersected with *W* to ensure that partially-updated cache lines are merged correctly [8].

to support two operations. The first one is to commit C_{pred} and C_{succ} in order. Note that this does not preclude the processor from *overlapping* the execution of C_{pred} and C_{succ}. Such overlapped execution improves performance, and can be managed by separately disambiguating the two chunks with per-chunk *R* and *W* signatures. However, if C_{pred} needs to be squashed, then we also squash C_{succ}.

The second operation is to update the *R* signature of C_{succ} correctly and in a timely manner, when there is data forwarding from a write in C_{pred} to a read in C_{succ}. In particular, the timing aspect of this operation is challenging, since the update to the *R* signature of C_{succ} may take a few cycles and occur after the data is consumed. This opens a window of vulnerability where a data collision between an external, committing chunk and C_{succ} could be missed: between the time a load in C_{succ} consumes the data and the time the *R* signature of C_{succ} is updated. Note that if C_{pred} has not yet committed, a data collision cannot be missed: the *W* signature of C_{pred} will flag the collision and both C_{pred} and C_{succ} will be squashed. However, if C_{pred} has committed and cleared its *W* signature, we can only rely on the *R* signature of C_{succ} to flag the collision. Consequently, the implementation must make sure that, by the time C_{pred} commits, its forwards to C_{succ} have been recorded in the *R* signature of C_{succ}.

CReq2 (single sequential order requirement) can be conservatively enforced with the combination of two operations: (i) *total order* of chunk commits and (ii) *atomic* chunk commit. To support the first one, we rely on the arbiter. Before a processor can commit a chunk, it asks permission to the arbiter. The arbiter ensures that chunks commit one at a time, without overlap. If no chunk is currently committing, the arbiter grants permission to the requester. To support atomic chunk commit, we disable access to all the memory locations that have been modified by the chunk, while the chunk is committing. No reads or writes from any processor to these locations in memory are allowed. When the committing chunk has made all its updates visible (e.g., by invalidating all the corresponding lines from all other caches), access to all these memory locations is re-enabled in one shot.

3.2.2. Advanced Design

To enforce *CReq2*, the naive design places two unnecessary constraints that limit parallelism: (i) chunk commits are completely serialized and (ii) access to the memory locations written by a committing chunk is disabled for the duration of the whole commit process. We can eliminate these constraints and still enforce *CReq2* with a simple solution: when a processor sends to the arbiter a permission-to-commit request for a chunk, it includes the chunk's *R* and *W* signatures.

To relax the first constraint, we examine the sufficient conditions for an implementation of SC at the memory access level. According to [1], the single sequential order requirement (*Req2* in Section 2.1) requires only that (i) writes to the same location be made visible to all processors in the same order (write serialization), and (ii) the value of a write not be returned by a read until the write becomes visible to all processors. Therefore, *Req2* puts constraints on the accesses to a single updated location. In a chunk environment, these constraints apply to all the locations updated by the committing chunk, namely those in its *W* signature. Consequently, it is safe to overlap the commits of two chunks with non-overlapping *W* signatures — unless we also want to relax the second constraint above (i.e., disabling access to the memory locations written by the committing chunk, for the duration of the commit), which we consider next.

To relax this second constraint, we proceed as follows. As a chunk commits, we re-enable access to individual lines gradually, as soon as they have been made visible to all other processors. Since these lines are now part of the committed state, they can be safely observed. Relaxing this constraint enhances parallelism, since it allows a currently-running chunk to read data from a currently-committing one sooner. Moreover, in a distributed-directory machine, it eliminates the need for an extra messaging step between the directories to synchronize the global end of commit.

However, there is one corner case that we must avoid because it causes two chunks to commit out of order and, if a third one observes it, we broke *CReq2*. This case occurs if chunk *C1* is committing, chunk *C2* reads a line committed by *C1*, and then *C2* starts committing as well and finishes before *C1*. One example is shown in Figure 4(b). In the figure, *C1* wrote *A* and *B* and is now committing. However, *A* and *B* are in different memory modules, and while *B* is quickly committed and access to it re-enabled, the commit signal has not yet reached *A*'s module (and so access to *A* is not disabled yet). *C2* reads the committed value of *B*, stores *C* and commits, completing before *C1*. This out-of-order commit is observed by *C3*, which reads the new *C* and the *old A* and commits — before receiving the incoming *W* signature of *C1*. We have violated SC.

A simple solution that guarantees avoiding this and similar violations is for the arbiter to deny a commit request from a chunk whose *R* signature overlaps with the *W* signature of any of the currently-committing chunks.

To summarize, in our design, the arbiter keeps the *W* signatures of all the currently-committing chunks. An incoming commit request includes a *R* and a *W* signature. The arbiter grants permission only if all its own *W* signatures have an empty intersection with the incoming *R* and *W* signature pair. This approach enables high parallelism because (i) multiple chunks can commit concurrently and (ii) a commit operation re-enables access to different memory locations as soon as possible to allow incoming reads to proceed. The latter is especially effective across directory modules in a distributed machine.

3.2.3. Overall *BulkSC* System

We propose to support bulk enforcement of SC as described with an architecture called *BulkSC*. *BulkSC* leverages a cache hierarchy with support for Bulk operations and a processor with efficient checkpointing. The memory subsystem is extended with an arbiter module. For generality, we focus on a system with a distributed directory protocol and a generic network. Figure 5 shows an overview of the architecture.

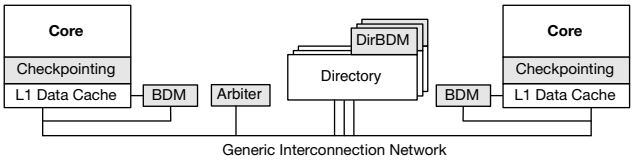


Figure 5. Overview of the *BulkSC* architecture.

All processors repeatedly (and only) execute chunks, separated by checkpoints. As a processor executes a chunk speculatively, it buffers the updates in the cache and generates a *R* and a *W* signature in the BDM. When chunk *i* completes, the processor sends a

request to commit to the arbiter with signatures R_i and W_i. The arbiter intersects R_i and W_i with the *W* signatures of all the currently-committing chunks. If all intersections are empty, W_i is saved in the arbiter and also forwarded to all interested directories for commit. Individual directories use a DirBDM module to perform signature expansion [8] on W_i to update their sharing state, and forward W_i to interested caches. The BDM in each cache uses W_i to perform bulk disambiguation and potentially squash local chunks. Memory accesses within a chunk are fully overlapped and reordered, and an individual processor can overlap the execution of multiple chunks.

3.3. Interaction with Explicit Synchronization

A machine with *BulkSC* runs code with explicit synchronization operations correctly. Such operations are executed inside chunks. While they have the usual semantics, they neither induce any fences nor constrain access reordering within the chunk in any way.

Figure 6 shows some examples with lock acquire and release. In Figure 6(a), acquire and release end up in the same chunk. Multiple processors may execute the critical section concurrently, each believing that it owns the lock. The first one to commit the chunk squashes the others. The longer the chunk is relative to the critical section, the higher the potential for hurting parallelism is — although correctness is not affected. Figure 6(b) shows a chunk that includes two critical sections. Again, this case may restrict parallelism but not affect correctness. Finally, Figure 6(c) shows a chunk that only contains the acquire. Multiple processors may enter the critical section believing they own the lock. However, the first one to commit squashes the others which, on retry, find the critical section busy.

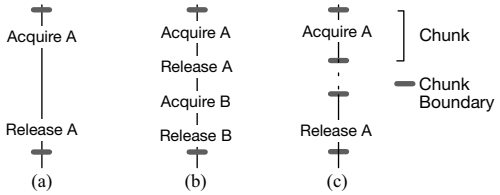


Figure 6. Interaction of *BulkSC* with explicit synchronization.

Similar examples can be constructed with other primitives. A worst case occurs when all processors but one are waiting on a synchronization event — e.g., when they are waiting on the processor that holds a lock, the processor that arrives last to a barrier, or the one that will set a flag. In this case, *BulkSC* guarantees that the key processor makes forward progress. To see why, note that the waiting processors are spinning on a variable. Committing a chunk that just reads a variable cannot squash another chunk. However, one could envision a scenario where the spin loop includes a write to variable *v*, and *v* (or another variable in the same memory line) is read by the key processor. In this case, the latter could be repeatedly squashed.

This problem is avoided by dynamically detecting when a chunk is being repeatedly squashed and then taking a measure to prevent future squashing. *BulkSC* includes two such measures: one for high performance that works in the common case, and one that is slow but guarantees progress. The first one involves exponentially decreasing the size of the chunk after each squash, thereby significantly increasing the chances that the chunk will commit. However, even reducing a chunk's size down to a single write does not guarantee forward progress. The second measure involves pre-arbitrating. Specifically, the processor first asks the arbiter permission to execute; once permission is granted, the processor executes while the arbiter rejects commit requests from other processors. After the arbiter receives the commit request from the first processor, the system returns to normal operation.

Finally, while chunks and transactions share some similarities, chunks are not programming constructs like transactions. Indeed, while transactions have static boundaries in the code, chunks are dynamically built by the hardware from the dynamic instruction stream. Therefore, they do not suffer the livelock problems pointed out by Blundell *et al*. [5].

4. BulkSC Architecture

We consider the three components of the *BulkSC* architecture: support in the processor and cache (Section 4.1), the arbiter module (Section 4.2), and directory modifications (Section 4.3).

4.1. Processor and Cache Architecture

4.1.1. Chunk Execution

Processors dynamically break the instruction stream into chunks, creating a checkpoint at the beginning of each chunk. As indicated before, within-chunk execution proceeds with all the memory access reordering and overlapping possible in uniprocessor code. Explicit synchronization instructions do not insert any fences or constrain access reordering in any way. In addition, a processor can have multiple chunks in progress, which can overlap their memory accesses. As chunks from other processors commit, they disambiguate their accesses against local chunks, which may lead to the squash and re-execution of a local chunk. Finally, when a chunk completes, it makes all its state visible with a commit.

This mode of execution is efficiently underpinned by the mechanisms of Bulk [8] and of checkpointed processors [3, 7, 10, 19, 20, 23, 24, 29]. Bulk enables the inexpensive recording of the addresses accessed by a chunk in a *R* and *W* signature in the BDM. Loads update the *R* signature when they bring data into the cache, while stores update the cache and the *W* signature when they reach the ROB head — even if there are other, in-progress stores. Forwarded loads also update the *R* signature. With such signatures, chunk commit involves sending the chunk's *R* and *W* signatures to the arbiter and, on positive reply, clearing them. Cross-chunk disambiguation involves intersecting the incoming *W* signature of a committing chunk against the local *R* and *W* signatures — and squashing and re-executing the chunk if the intersection is not empty. Chunk rollback leverages the mechanisms of checkpointed processors: a register checkpoint is restored and all the speculative state generated by the chunk is efficiently discarded from the cache.

With this design, there is no need to snoop the load-store queue to enforce consistency, or to have a history buffer as in [15]. An SC violation is detected when a bulk disambiguation operation detects a non-empty intersection between two signatures.

Moreover, there is no need to watch for cache displacements to enforce consistency. Clean lines can be displaced from the cache, since the *R* signature records them, while the BDM prevents the displacement of speculatively written lines until commit [8]. Thanks to the BDM, the cache tag and data array are unmodified; they do not know if a given line is speculative or what chunk it belongs to.

4.1.2. Chunk Duration and Multiple-Chunk Support

The processor uses instruction count to decide when to start a new chunk. While chunks should be large enough to amortize the commit cost, very large chunks could suffer more conflicts. In practice, performance is fairly insensitive to chunk size, and we use chunks of ≈1,000 instructions. However, if the processor supports checkpoint-based optimizations, such as resource recycling [3, 23] or memory latency tolerance [7, 20, 29], it would make sense to use their checkpoint-triggering events as chunk boundaries. Finally, a chunk also finishes when its data is about to overflow a cache set.

A processor can have multiple chunks in progress. For this, it leverages Bulk's support for multiple pairs of signatures and a checkpointed processor's ability to have multiple outstanding checkpoints. When the processor decides that the next instruction to rename starts a new chunk, it creates a new checkpoint, allocates a new pair of *R* and *W* signatures in the BDM, and increments a set of bits called *Chunk ID*. The latter are issued by the processor along with every memory address to the BDM. They identify the signature to update.

Instructions from multiple local chunks can execute concurrently and their memory accesses can be overlapped and reordered, since they update different signatures. An incoming *W* signature performs disambiguation against all the local signature pairs and, if a chunk needs to be squashed, all its successors are also squashed.

Before a chunk can start the commit process, it ensures that all its forwards to local successor chunks have updated the successors's *R* signatures. As indicated in Section 3.2.1, this is required to close a window of vulnerability due to the lag in updating signatures. In our design, on any load forwarding, we log an entry in a buffer until the corresponding *R* signature is updated. Moreover, a completed chunk cannot initiate its commit arbitration until it finds the buffer empty. While a chunk arbitrates for commit or commits, its local successor chunks can continue execution. Local chunks must request and obtain permission to commit in strict sequential order. After that, their commits can overlap.

4.1.3. I/O

I/O and other uncached operations cannot be executed speculatively or generally overlapped with other memory accesses. When one such instruction is renamed, the processor stalls until the current chunk completes its commit — checking this event may require inspecting the arbiter. Then, the operation is fully performed. Finally, a new chunk is started. To support these steps, we reuse the relevant mechanisms that exist to deal with I/O in checkpointed processors.

4.1.4. Summary: Simplicity and Performance

We claim that *BulkSC*'s SC implementation is simple because it largely *decouples* memory consistency enforcement from processor structures. Specifically, there is no need to perform associative lookups of sizable structures in the processor, or to interact in a tightly-coupled manner with key structures such as the load-store queue, ROB, register file, or map table.

This is accomplished by leveraging Bulk and checkpointed processors. With Bulk, detection of SC violations is performed with simple signature operations outside the processor core. Additionally, caches are oblivious of what data is speculative (their tag and data arrays are unmodified), and do not need to watch for displacements to enforce consistency. Checkpointing provides a non-intrusive way to recover from consistency-violating chunks.

In our opinion, this decoupling enables designers to conceive both simple and aggressive (e.g., CFP [29], CAVA/Clear [7, 20], CPR [3] or Kilo-instruction [10]) processors with much less concern for memory consistency issues.

BulkSC delivers high performance by allowing any reordering and overlapping of memory accesses within a chunk. Explicit synchronization instructions induce no fence or reordering constraint. Moreover, a processor does not stall on chunk transitions, and it can overlap and reorder memory operations from different chunks. Finally, arbitration for chunk commit is quick. It is not a bottleneck because it only requires quick signature intersections in the arbiter. In the meantime, the processor continues executing.

4.2. Arbiter Module

4.2.1. Baseline Design

The arbiter is a simple state machine whose role is to enforce the minimum serialization requirements of chunk commit. The arbiter stores the *W* signatures of all the currently-committing chunks. It receives permission-to-commit requests from processors that include the *R* and *W* signatures of a chunk. The arbiter takes each of the *W* signatures in its list and intersects them with the incoming *R* and *W* signatures. If any intersection is not empty, permission is denied; otherwise, it is granted, and the incoming *W* signature is added to the list. Since this process is very fast, the arbiter is not a bottleneck in a modest-sized machine. A processor whose request is denied will later retry.

Figure 7(a) shows the complete commit process. Processor *A* sends a permission-to-commit message to the arbiter, together with *R* and *W* signatures (1). Based on the process just described, the arbiter decides if commit is allowed. It then sends the outcome to the processor (2) and, if the outcome was positive, it forwards the *W* signature to the relevant directories (2). Each directory processes *W* (Section 4.3) and forwards it to the relevant processors (3), collects operation-completion acknowledgements (4) and sends a completion message to the arbiter (5). When the arbiter receives all the acknowledgements, it removes the *W* signature from its list.

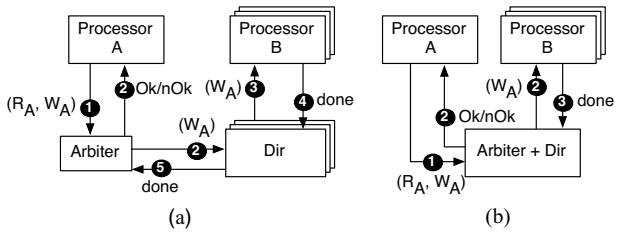


Figure 7. Commit process with separate (a) and combined (b) arbiter and directory.

In a small machine like a few-core chip multiprocessor, there may be a single directory. In this case, the arbiter can be combined with it. The resulting commit transaction is shown in Figure 7(b).

4.2.2. *RSig* Commit Bandwidth Optimization

The list of *W* signatures in the arbiter of a modest-sized machine is frequently empty. This is because the commit process is fast and — as we will see in Section 5 — chunks that only write processor-private data have an empty *W* signature.

When the arbiter's *W* signature list is empty, the arbiter has no use for the *R* signature included in a permission-to-commit request. We can save network bandwidth if we do not include the *R* signature in the message. Consequently, we improve the commit design by always sending only the *W* signature in the permission-to-commit request. In the frequent case when the arbiter's list is empty, the arbiter grants permission immediately; otherwise, it requests the *R* signature from the processor and proceeds as before. We call this optimization *RSig* and include it in the baseline *BulkSC* system.

4.2.3. Distributed Arbiter

In large machines, the arbiter may be a bottleneck. To avoid this case, we distribute the arbiter into multiple modules, each managing a range of addresses. An arbiter now only receives commit requests from chunks that have accessed its address range (plus potentially other ranges as well). To commit a chunk that only accessed a single address range, a processor only needs to communicate with one arbiter. For chunks that have accessed multiple ranges, multiple arbiters need to be involved. In this case, each arbiter will make a decision based on the partial information that it has — the *W* signatures of the committing chunks that have written its address range. We then need an extra arbiter module that coordinates the whole transaction. We call this module *Global Arbiter* or *G-arbiter*.

Figure 8 shows the two possible types of commit transactions, in a machine where we have distributed the arbiter with the distributed directory. Figure 8(a) shows the common case of a commit that involves a single arbiter. The processor knows from the signatures which arbiter to contact. Figure 8(b) shows the case when multiple arbiters are involved. In this case, the processor sends the signatures to the G-arbiter (1), which forwards them to the relevant arbiters (2). The arbiters check their *W* list and send their responses to the G-arbiter (3), which combines them and informs all parties (4). The transaction then proceeds as usual.

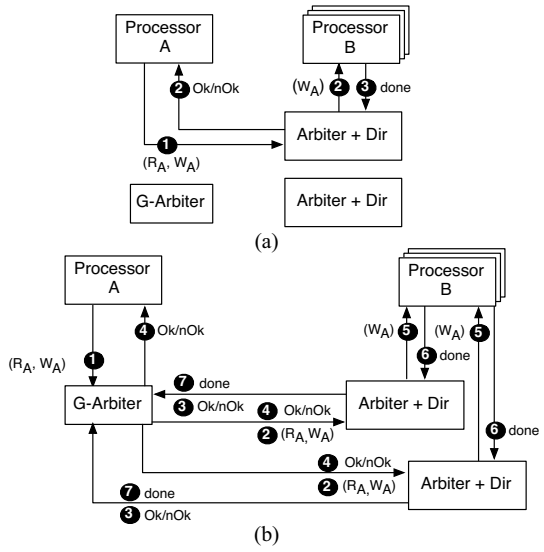


Figure 8. Distributed arbiter with a commit that involves a single arbiter (a) or multiple (b).

When the G-arbiter is used, the commit transaction has longer latency and more messages. We can speed up some transactions by storing in the G-arbiter the *W* signatures of all the currently-committing chunks whose requests went through the G-arbiter. With this, we are replicating information that is already present in some arbiters, but it may help speed up transactions that are denied. Indeed, in Figure 8(b), when the G-arbiter receives message (1), it checks its *W* list for collisions. If it finds one, it immediately denies permission.

4.3. Directory Module

BulkSC does not require a machine with a broadcast link to work. For generality, this paper presents a design with distributed memory and directory. However, we need to extend the directory to work with the inexact information of signatures. This is done by enhancing each directory module with a module called *DirBDM* that supports basic bulk operations (Figure 5). When a directory module receives the *W* signature of a committing chunk, the DirBDM performs three operations: (i) expand the signature into its component addresses and update the directory state, (ii) based on the directory state, forward *W* to the relevant caches for address disambiguation, and (iii) conservatively disable access to the directory entries of all the lines written by the committing chunk in this module, until the new values of the lines are visible to all processors — i.e., until the old values of the lines have been invalidated from all caches.

Operation (iii) is conservative; we could have re-enabled access to individual lines in the directory module progressively, as they get invalidated from all caches. However, we choose this implementation for simplicity. Still, different directory modules re-enable access at different times.

In our discussion, we use a full bit-vector directory [22] for simplicity. Directory state can be updated when a chunk commits, when a non-speculative dirty line is written back, or when the directory receives a demand cache miss from a processor. The latter are always read requests — even in the case of a write miss — and the directory adds the requester processor as a sharer. The reason is that, because the access is speculative, the directory cannot mark the requester processor as keeping an updated copy of the line.

4.3.1. Signature Expansion

On reception of a *W* signature, the DirBDM performs a signature-expansion bulk operation (Section 2.2) to determine what entries in the directory structure may have their addresses encoded in the signature. For each of these entries, it checks the state and, based on it, (i) compiles a list of processors to which *W* will be forwarded for bulk disambiguation and (ii) updates the entry's state. The list of processors that should receive *W* is called the Invalidation List.

A key challenge is that the signature expansion of *W* may produce the address of lines that have not been written. Fortunately, a careful analysis of all possible cases ensures that the resulting directory actions never lead to incorrect execution. To see why, consider the four possible states that one of the selected directory entries can be at (Table 1). In all cases, we may be looking at a line that the chunk did not actually write.

In the table, cases 1 and 3 from the top are clearly false positives: if the committing chunk had written the line, its processor would have accessed the line and be recorded in the bit vector as a sharer, already. Therefore, no action is taken. Case 4 requires no action even if it is not a false positive. Case 2 requires marking the committing processor as keeping the only copy of the line, in state dirty, and adding the rest of current sharer processors to the Invalidation

Current Entry State		Action	Notes
Dirty Bit Set?	Committing Proc is in Bit Vector?		
No	No	Do nothing	False positive. Committing proc should have accessed the data and be in bit vector already
No	Yes	1) Add sharer procs to Invalidation List 2) Reset rest of bit vector 3) Set Dirty bit	Committing proc becomes the owner
Yes	No	Do nothing	False positive. Committing proc should have accessed the data and be in bit vector already
Yes	Yes	Do nothing	Committing proc already owner

Table 1. Possible states of a directory entry selected after signature expansion and action taken.

List. If this is a false positive, we are incorrectly changing the directory state and maybe forwarding *W* to incorrect processors. The latter can at most cause unnecessary (and rare) chunk squashes — it cannot lead to incorrect execution; the former sets the coherence protocol to a state that many existing cache coherence protocols are already equipped to handle gracefully — consequently, it is not hard to modify the protocol to support it. Specifically, it is the same state that occurs when, in a MESI protocol [26], a processor reads a line in Exclusive mode and later displaces it from its cache silently. The directory thinks that the processor owns the line, but the processor does not have it.

In our protocol, at the next cache miss on the line by a processor, the directory will ask the "false owner" for a writeback. The latter will respond saying it does not have a dirty copy. The directory will then provide the line from memory, and change the directory state appropriately.

4.3.2. Disabling Access from Incoming Reads

As per Section 3.2.2, *CReq2* (single sequential order requirement) requires that no processor use the directory to see the new value of a line from the committing chunk, before all processors can see the new value. As indicated above, our conservative implementation disables reads to any line updated by the committing chunk in this module, from the time the directory module receives the *W* signature from the arbiter (Message (2) in Figure 7(a)) until it receives the "done" messages from all caches in the Invalidation List (Messages (4)). This is easily done with bulk operations. The DirBDM intercepts all incoming reads and applies the *membership* bulk operation (Section 2.2) to them, to see if they belong to *W*. If they do, they are bounced.

4.3.3. Directory Caching

In *BulkSC*, using directory caches [16] is preferred over full-mapped directories because they limit the number of false positives by construction. With a directory cache, the DirBDM uses signature expansion (Section 2.2) on incoming *W* signatures with a different decode (δ) function than for the caches — since directories have different size and associativity.

However, directory caches suffer entry displacements. In conventional protocols, a displacement triggers the invalidation of the line from all caches and, for a dirty line, a writeback. In *BulkSC*, a conservative approach is to additionally squash all the currently-running chunks that may have accessed the line. Consequently, when the directory displaces an entry, it builds its address into a signature and sends it to all sharer caches for bulk disambiguation with their *R* and *W* signatures. This operation may squash chunks, and will invalidate (and if needed write back) any cached copies of the line.

This protocol works correctly for a chunk that updated the displaced line and is currently committing. Since the chunk has already cleared its signatures, disambiguation will not squash it. Moreover, since the line is dirty non-speculative in the processor's cache, the protocol will safely write back the line to memory.

5. Leveraging Private Data

Accesses to private data are not subject to consistency constraints. Consequently, they do not need to be considered when disambiguating chunks or arbitrating for commits — they can be removed from a chunk's *W* signature.

Removing writes to private data from *W* also has the substantial benefit of reducing signature pollution. This decreases aliasing and false positives, leading to fewer unnecessary cache invalidations and chunk squashes. Another benefit is that the resulting *W* is often empty. This reduces the number of entries in the arbiter's list of *W* signatures — since a permission-to-commit request with a zero *W* does not update the list. As a result, such a list is more often empty. This has enabled the *RSig* commit bandwidth optimization of Section 4.2.2.

To handle private data, we propose to include in the BDM an additional signature per running chunk called W_{priv}. Writes to private data update W_{priv} rather than *W*. W_{priv} is used neither for bulk disambiguation nor for commit arbitration. In the following, we present two schemes that use W_{priv} to leverage either statically-private data or dynamically-private data. The latter is data that, while perhaps declared shared, is accessed by only one processor for a period of time. These two schemes use W_{priv} differently.

5.1. Leveraging Statically-Private Data

This approach relies on the software to convey to the hardware what data structures or memory regions are private. A simple implementation of this approach is to have a page-level attribute checked at address-translation time that indicates whether a page contains private or shared data. On a read to private data, the *R* signature is not updated, thereby avoiding polluting *R*; on a write to private data, W_{priv} is updated rather than *W*.

To commit a chunk, the processor only sends *W* to the arbiter. Once commit permission is granted, the processor sends W_{priv} directly to the directory for signature expansion, so that private data is kept *coherent*. Coherence of private data is necessary because threads can migrate between processors, taking their private data to multiple processors. With this approach, we have divided the address space into a section where SC is enforced and another where it is not.

5.2. Leveraging Dynamically-Private Data

In many cases, a processor *P* updates a variable *v* in multiple chunks without any intervening access to *v* from other processors. Using our protocol of Section 4, every first write of *P* to *v* in a chunk

would require the update of *W* and the writeback of *v*'s line to memory. The latter is necessary because *v*'s line is in state dirty non-speculative before the write.

We want to optimize this common case by (i) skipping the writeback of *v*'s line to memory at every chunk and (ii) not polluting *W* with the writes to *v*. Our optimization, intuitively, is to update *v* keeping its line in dirty *non-speculative* state in the cache. If, by the time the chunk completes, no external event required the old value of *v*, the processor commits the chunk without informing the arbiter or directory that the chunk updated *v* — more specifically, the processor sends to the arbiter a *W* signature that lacks the writes to *v*.

To support this, we make two changes to *BulkSC*. First, every time that a processor writes to a line that is dirty non-speculative in its cache, the hardware updates W_{priv} rather than *W*, and does not write the line back to memory. These lines can be easily identified by the hardware: they have the Dirty bit set and their address is not in *W*. With this change, the commit process will not know about the updates to these lines.

The second change is that, the first time that these lines are updated in a chunk, the hardware also saves the version of the line before the update in a small *Private Buffer* in the BDM, while the updated version of the line is kept in the cache. The hardware easily identifies that this is the first update: the Dirty bit set and the line's address is neither in *W* nor in W_{priv}. We save the old value of the line in this buffer in case it is later required. This buffer can hold $\approx$24 lines and is not in any critical path.

With this support, when a chunk is granted permission to commit based on its *W*, the hardware clears the Private Buffer and W_{priv}. We have skipped the writeback of the lines in the buffer.

There are, however, two cases when the old value of the line is required. The first one is when the chunk gets squashed in a bulk disambiguation operation. In this case, the lines in the Private Buffer are copied back to the cache — and the Private Buffer and W_{priv} are cleared.

The second case is when our predicted private pattern stops working, and another processor requests a line that is in the Private Buffer. This is detected by the BDM, which checks every external access to the cache for membership ($\in$) in W_{priv}. This is a fast bulk operation (Section 2.2). In the (unusual) case of a non-empty result, the Private Buffer is checked. If the line is found, it is supplied from there rather than from the cache, and the address is added to *W*. Intuitively, we have to provide the old copy of the line and "add back" the address to *W*. Similarly, if a line overflows the Private Buffer, it is written back to memory and its address is added to *W*.

6. Discussion

Past approaches to supporting high-performance SC (Section 2.1) add mechanisms local to the processor to enforce consistency; *BulkSC*, instead, relies on an external arbiter. We argue, however, that this limitation is outweighed by *BulkSC*'s advantage of simpler processor hardware — coming from decoupling memory consistency enforcement from processor microarchitecture and eliminating associative lookups (Section 4.1.4).

In addition, the amount of commit bandwidth required by *BulkSC* is very small. There are three reasons for this. First, since chunks are large, the few messages needed per individual commit are amortized over many instructions. Second, we have presented optimizations to reduce the commit bandwidth requirements due to *R* (Section 4.2.2) and *W* (Section 5) signatures; these optimizations are very effective, as we will show later. Finally, in large machines with a distributed arbiter, if there is data locality, commits only access a local arbiter.

BulkSC's scalability is affected by two main factors: the ability to provide scalable arbitration for chunk commit, and whether the superset encoding used by signatures limits scalability in any way. To address the first factor, we have presented commit bandwidth optimizations, and a distributed arbiter design that scales as long as there is data locality. Superset encoding could hurt scalability if the longer chunks needed to tolerate longer machine latencies ended up creating much address aliasing. In practice, there is a large unexplored design space of signature size and encoding. Superset encoding could also hurt scalability if it induced an inordinate number of bulk disambiguations as machine size increases. Fortunately, Section 4.3.1 showed that signature expansion can leverage the directory state to improve the precision of sharer information substantially.

Finally, write misses are a common source of stalls in multiprocessors. The effect of these misses on performance is more pronounced in stricter memory consistency models. In *BulkSC*, writes are naturally stall-free. Specifically, writes retire from the head of the reorder buffer even if the line is not in the cache — although the line has to be received before the chunk commits. Moreover, writes do not wait for coherence permissions because such permissions are implictly obtained when the chunk is allowed to commit.

7. Evaluation

7.1. Experimental Setup

We evaluate *BulkSC* using the SESC [28] cycle-accurate execution-driven simulator with detailed models for processor, memory subsystems and interconnect. For comparison, we implement SC with hardware prefetching for reads and exclusive prefetching for writes [12]. In addition, we implement RC with speculative execution across fences and hardware exclusive prefetching for writes. The machine modeled is an 8-processor CMP with a single directory. Table 2 shows the configurations used.

We use 11 applications from SPLASH-2 (all but Volrend, which cannot run in our simulator), and the commercial applications SPECjbb2000 and SPECweb2005. SPLASH-2 applications run to completion. The commercial codes are run using the Simics full-system simulator as a front-end to our SESC simulator. SPECjbb2000 is configured with 8 warehouses and SPECweb2005 with the e-commerce workload. Both run for over 1B instructions after initialization.

We evaluate the *BulkSC* configurations of Table 2: BSC_{base} is the basic design of Section 4; BSC_{dypvt} is BSC_{base} plus the dynamically-private data optimization of Section 5.2; BSC_{stpvt} is BSC_{base} plus the statically-private data optimization of Section 5.1; and BSC_{exact} is BSC_{base} with an alias-free signature. We also compare them to RC, SC and SC++ [15].

7.2. Performance

Figure 9 compares the performance of BSC_{base}, BSC_{dypvt}, BSC_{exact}, and BSC_{stpvt} to that of SC, RC and SC++. In the figure, *SP2-G.M.* is the geometric mean of SPLASH-2. Our preferred configuration, BSC_{dypvt}, performs about as well as RC and SC++ for practically all applications. Consequently, we argue that BSC_{dypvt}

Processor and Memory Subsystem		*BulkSC*	Configurations Used	
Cores: 8 in a CMP Frequency: 5.0 GHz Fetch/issue/comm width: 6/4/5 I-window/ROB size: 80/176 LdSt/Int/FP units: 3/3/2 Ld/St queue entries: 56/56 Int/FP registers: 176/90 Branch penalty: 17 cyc (min)	Private writeback D-L1: size/assoc/line: 32KB/4-way/32B Round trip: 2 cycles MSHRs: 8 entries Shared L2: size/assoc/line: 8MB/8-way/32B Round trip: 13 cycles MSHRs: 32 entries Mem roundtrip: 300 cyc	Signature: Size: 2 Kbits Organization: Like in [8] # Chunks/Proc: 2 Chunk Size: 1000 inst. Commit Arb. Lat.: 30 cyc Max. Simul. Commits: 8 # of Arbiters: 1 # of Directories: 1	BSC_{base} BSC_{dypvt} BSC_{stpvt} BSC_{exact} SC RC SC++	Basic *BulkSC* (Section 4) BSC_{base} + dyn. priv. data opt (Section 5.2) BSC_{base} + static priv. data opt (Section 5.1) BSC_{dypvt} + "magic" alias-free signature Includes prefetching for reads + exclusive prefetching for writes Includes speculative execution across fences + exclusive prefetching for writes Model from [15] w/ 2K-entry SHiQ

Table 2. Simulated system configurations. All latencies correspond to an unloaded machine.

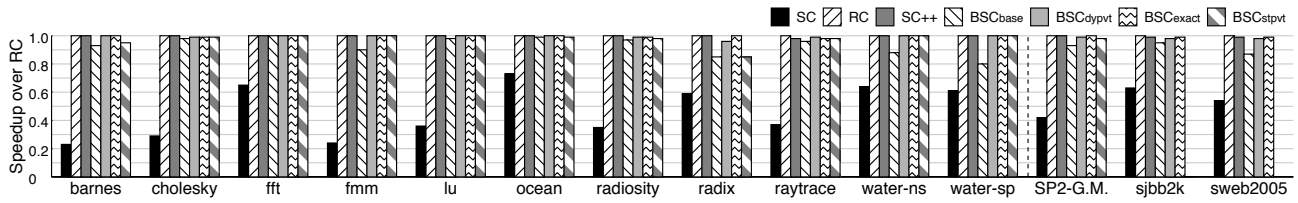


Figure 9. Performance of several *BulkSC* configurations, SC, RC and SC++, all normalized to RC.

is the most attractive design of the three, as it is simple to implement and still supports SC. The only exception is radix, which has frequent chunk conflicts due to aliasing in the signature. The performance difference between RC and SC is large, and in line with [25].

Comparing BSC_{base} and BSC_{dypvt}, we see the impact of the dynamically-private data optimization. It improves performance by 6% in SPLASH-2, 3% in SPECjbb and 11% in SPECweb. Most of the gains come from reducing the pollution in the *W* signature, leading to fewer line invalidations and chunk squashes. The small difference between BSC_{exact} and BSC_{dypvt} shows that the dynamically-private data optimization reduces aliasing enough to make BSC_{dypvt} behave almost as if it had an ideal signature.

For BSC_{stpvt}, we consider all stack references as private and everything else as shared. Unfortunately, we can only apply it to SPLASH-2 applications because of limitations in our simulation infrastructure. As Figure 9 shows, BSC_{stpvt} improves over BSC_{base} by a geometric mean of 5%, leaving BSC_{stpvt} within 2% of the performance of BSC_{dypvt}. The only application with no noticeable benefit is radix, which has very few stack references.

Figure 10 shows the performance of BSC_{dypvt} with chunks of *1000*, *2000* and *4000* instructions. *4000-exact* is BSC_{exact} with 4000-instruction chunks. We see that for a few SPLASH-2 applications and for the commercial ones, performance degrades as chunk size increases. Comparing *4000* to *4000-exact*, however, we see that most of the degradation comes from increased aliasing in the signature, rather than from real data sharing between the chunks.

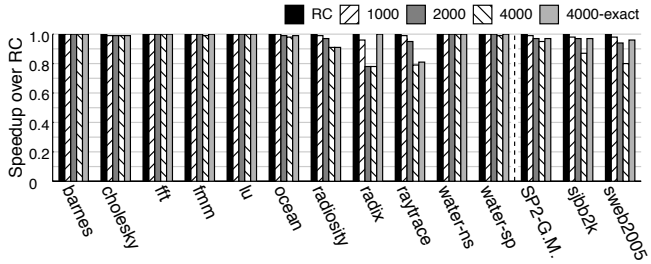


Figure 10. BSC_{dypvt} performance with different chunk sizes.

7.3. General Characterization of *BulkSC*

Table 3 characterizes *BulkSC*. Unless otherwise indicated, the data is for BSC_{dypvt}. We start with the *Squashed Instructions* columns. In BSC_{exact}, squashes are due to false and true sharing, while in BSC_{dypvt} and BSC_{base}, squashes also occur due to aliasing in the signatures. While the wasted work in BSC_{base} is 8-10% of the instructions, in BSC_{dypvt} it is only 1-2%, very close to that in BSC_{exact}. This is due to reduced aliasing.

The *Average Set Sizes* columns show the number of cache line addresses encoded in the BSC_{dypvt} signatures. We see that *Priv. Write* has many more addresses than *Write*. The *Priv. Write* data shows that a Private Buffer of ≈24 entries is typically enough.

The *Spec Line Displacements* columns show how often a line speculatively read or written is displaced from the cache per 100k commits. These events are very rare in SPLASH-2. They are less rare in the commercial applications, where they can be as high as 1 per 10 commits for the read set. Note, however, that in *BulkSC*, displacements of data read *do not* cause squashes, unlike in SC++. The column *Data from Priv. Buff.* shows how frequently the Private Buffer has to provide data. As expected, this number is very low — 6-9 times per 1k commits on average.

When a processor receives the *W* signature of a committing chunk, it uses it to invalidate lines in its cache. Due to aliasing in the signatures, it may invalidate more lines than necessary. The *Extra Cache Invs.* column quantifies these unnecessary invalidates per 1k commits. This number is very low and unlikely to affect performance, since these lines are likely to be refetched from L2.

7.4. Commit and Coherence Operations

Table 4 characterizes the commit process and the coherence operations in BSC_{dypvt}. The *Signature Expansion* columns show data on the expansion of *W* signatures in the directory. During expansion, directory entries are looked up to see if an action is necessary. The *Lookups per Commit* column shows the number of entries looked-up per commit. Since the Write Set sizes are small, this is a small number (7 in SPLASH-2 and 4 in the commercial applications). The

Appl.	Squashed Instructions (%)			Average Set Sizes in BSC_{dypvt} (Cache Lines)			Spec. Line Displacements (Per 100k Commits)		Data from Priv. Buff. (Per 1k Comm.)	# of Extra Cache Invs. (Per 1k Comm.)
	BSC_{exact}	BSC_{dypvt}	BSC_{base}	Read	Write	Priv. Write	Write Set	Read Set		
barnes	0.01	0.03	6.27	22.6	0.1	11.9	0.3	209.0	0.1	0.1
cholesky	0.04	0.05	2.18	42.0	0.9	11.6	0.0	57.2	1.0	0.2
fft	0.01	1.37	2.93	33.4	3.3	22.7	0.0	0.0	0.1	2.0
fmm	0.00	0.11	6.99	33.8	0.2	6.2	0.0	241.0	0.2	0.5
lu	0.00	0.00	3.29	15.9	0.1	10.8	0.0	0.9	0.0	0.0
ocean	0.35	0.92	2.14	45.3	6.7	8.4	0.0	1206.7	4.9	4.3
radiosity	0.98	1.04	4.25	28.7	0.5	15.2	0.0	50.7	29.9	28.8
radix	0.01	10.89	30.75	14.9	5.2	14.4	0.0	375.6	0.1	1760.0
raytrace	2.71	2.92	8.48	40.2	0.8	12.7	0.0	98.9	30.0	84.3
water-ns	0.03	0.07	12.67	20.2	0.1	16.3	0.0	2.7	0.3	1.9
water-sp	0.06	0.09	10.23	22.2	0.1	17.0	0.0	152.6	0.4	1.4
SP2-AVG	0.38	1.59	8.20	29.0	1.62	13.4	0.0	217.7	6.1	171.2
sjbb2k	0.45	1.11	10.33	43.6	3.56	19.2	1.8	6838.4	6.7	2.9
sweb2005	0.23	0.88	9.97	61.1	3.76	21.5	0.0	10502.5	8.7	4.1

Table 3. Characterization of *BulkSC*. Unless otherwise indicated, the data corresponds to BSC_{dypvt}

Appl.	Signature Expansion in Directory				Arbiter			*R* Sig. Required (% Commits)
	Lookups per Commit	Unnecessary Lookups (%)	Unnecessary Updates (%)	Nodes per *W* Sig.	# of Pend. *W* Sigs.	Non-Empty *W* List (% Time)	Empty *W* Sig. (% Commits)	
barnes	0.1	12.7	0.3	0.08	0.09	8.2	95.3	3.9
cholesky	1.2	27.7	0.0	0.18	0.03	2.9	98.1	1.1
fft	22.1	85.0	0.3	0.01	0.10	9.4	90.9	1.2
fmm	0.7	78.0	1.0	0.08	0.03	3.0	98.2	1.2
lu	0.1	16.7	0.0	0.01	0.06	5.7	96.8	2.7
ocean	9.5	29.9	0.4	0.05	0.53	40.0	55.8	13.6
radiosity	0.6	23.2	0.5	1.15	0.09	8.5	95.2	4.0
radix	37.8	86.2	0.4	1.10	0.56	49.3	32.9	15.5
raytrace	0.8	6.2	0.4	0.95	0.22	20.6	84.9	8.6
water-ns	0.2	42.0	0.7	0.74	0.02	1.4	99.2	0.7
water-sp	0.0	36.1	4.6	1.12	0.01	0.5	99.7	0.2
SP2-AVG	6.7	40.3	0.8	0.50	0.16	13.6	86.1	4.8
sjbb2k	4.0	10.1	0.1	0.06	0.54	46.1	46.9	17.8
sweb2005	4.5	17.0	0.2	0.09	0.65	51.7	49.5	28.1

Table 4. Characterization of the commit process and coherence operations in BSC_{dypvt}.

Unnecessary Lookups column is the fraction of these lookups performed due to aliasing. In SPLASH-2, about 40% of the lookups are unnecessary, while in the commercial applications only 10-17% are. The *Unnecessary Updates* column shows how many directory entries are unnecessarily updated due to aliasing. This number is negligible (0.1-0.8%). Finally, the *Nodes per W Sig.* column shows how many nodes receive the *W* signature from the directory. On average, a commit sends *W* to less than one node — often, the chunk has only written to data that is not cached anywhere else.

The next few columns characterize the arbiter. *Pend. W Sigs.* is the number of *W* signatures in the arbiter at a time, while *Non-Empty W List* is the percentage of time the arbiter has a non-empty *W* list. These numbers show that the arbiter is not highly utilized. *Empty W Sig.* shows how many commits have empty *W* signatures, due to accessing only private data. This number is 86% in SPLASH-2 and 47-50% in the commercial codes. Thanks to them, the arbiter's *W* list is often empty. Finally, *R Sig. Required* shows the fraction of commits that need the *R* signature to arbitrate. The resulting low number — 5% for SPLASH-2 and 18-28% for the commercial codes — shows that the *RSig* optimization works very well.

Figure 11 shows the interconnection network traffic in bytes, normalized to RC. We show RC (*R*), BSC_{exact} (*E*), BSC_{dypvt} without the *RSig* commit bandwidth optimization (*N*) and BSC_{dypvt} (*B*). The traffic is due to reads and writes (*Rd/Wr*), *R* signatures (*RdSig*), *W* signatures (*WrSig*), invalidations (*Inv*), and other messages.

This figure shows that the total bandwidth overhead of BSC_{dypvt} (*B*) compared to RC (*R*) is around 5-13% on average. This is very tolerable. The overhead is due to the transfer of signatures (*WrSig* and *RdSig*) and to extra data fetches after squashes (increase in *Rd/Wr*). The difference between *N* and *B* shows the effect of the *RSig* commit bandwidth optimization. We see that it has a significant impact. With this optimization, *RdSig* practically disappears from the *B* bars. Note that, without this optimization, *RdSig* is very significant in fmm and water-sp. These two applications have few misses and, consequently, signatures account for much of the traffic. Finally, the difference between *E* and *N* shows the modest effect of aliasing on traffic.

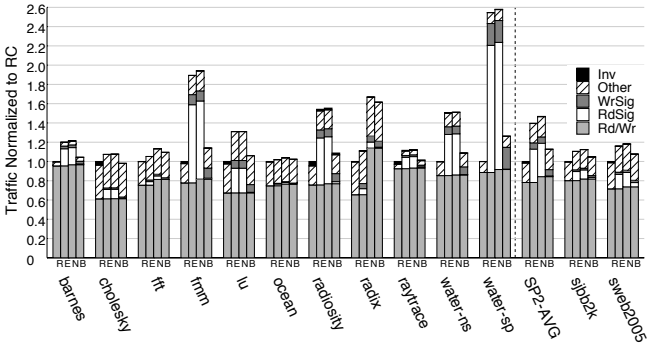


Figure 11. Traffic normalized to RC. *R*, *E*, *N*, *B* refer to RC, BSC_{exact}, BSC_{dypvt} without the *RSig* optimization, and BSC_{dypvt}, respectively.

8. Other Related Work

There is extensive literature on relaxed memory consistency models (e.g., [2, 11, 13]). Discussing it is outside our scope.

Chunks are similar to tasks in TLS or transactions in TM in that they execute speculatively and commit as a unit. In particular, an environment with transactions all the time such as TCC [17] is related to *BulkSC* where processors execute chunks all the time. However, the environments differ in that while tasks and transactions are statically specified in the code, chunks are created dynamically by the hardware. Still, *BulkSC* may be a convenient building block for TM and TLS because it will support SC ordering for the entire program, including transactional with respect to non-transactional code.

Concurrently with our work, Chafi *et al*. [9] have proposed an arbiter-based memory subsystem with directories for TM.

Also concurrently with our work, Wenisch *et al*. [30] have proposed Atomic Sequence Ordering (ASO) and a scalable store buffer design to enable store-wait-free multiprocessors. ASO, like BulkSC, makes a group of memory operations appear atomic to the rest of the system to avoid consistency violations. Our approach in BulkSC is to build the whole memory consistency enforcement based only on coarse-grain operation. In BulkSC, stores are also wait-free because they retire speculatively before the chunk commits.

9. Conclusion

This paper presented *Bulk enforcement of SC* or *BulkSC*, a novel implementation of SC that is simple to implement and delivers performance comparable to RC. The idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. Then, the hardware enforces SC at the coarse grain of chunks rather than at the fine grain of individual memory accesses. Enforcing SC at chunk granularity can be realized with simple hardware and delivers high performance. Moreover, to the program, it appears as providing SC at the memory access level.

BulkSC uses mechanisms from Bulk and checkpointed processors to largely decouple memory consistency enforcement from processor structures. There is no need to perform associative lookups or to interact in a tightly-coupled manner with key processor structures. Cache tag and data arrays remain unmodified, are oblivious of what data is speculative, and do not need to watch for displacements to enforce consistency. Overall, we believe that this enables designers to conceive processors with much less concern for memory consistency issues. Finally, *BulkSC* delivers high performance by allowing memory access reordering and overlapping within chunks and across chunks.

We also described a system architecture that supports *BulkSC* with a distributed directory and a generic network. We showed that *BulkSC* delivers performance comparable to RC. Moreover, it only increases the network bandwidth requirements by 5-13% on average over RC, mostly due to signature transfers and chunk squashes.

Acknowledgements

We thank the anonymous reviewers and the I-ACOMA group for their comments. Special thanks go to Brian Greskamp and José Martínez for their feedback. We thank Karin Strauss for the initial suggestion of using bulk operations for consistency enforcement.

References

[1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[2] S. V. Adve and M. Hill, "Weak Ordering - A New Definition," in *Inter. Symp. on Computer Architecture*, May 1990.

[3] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Inter. Symp. on Microarchitecture*, November 2003.

[4] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 11, July 1970.

[5] C. Blundell, E. Lewis, and M. Martin, "Subtleties of Transactional Memory Atomicity Semantics," *Computer Architecture Letters*, November 2006.

[6] H. Cain and M. Lipasti, "Memory Ordering: A Value-Based Approach," in *Inter. Symp. on Computer Architecture*, June 2004.

[7] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Using Checkpoint-Assisted Value Prediction to Hide L2 Misses," *ACM Transactions on Architecture and Code Optimization*, June 2006.

[8] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Inter. Symp. on Computer Architecture*, June 2006.

[9] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory," in *Inter. Symp. on High Performance Computer Architecture*, February 2007.

[10] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," in *Inter. Symp. on High Performance Computer Architecture*, February 2004.

[11] M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," in *Inter. Symp. on Comp. Architecture*, June 1986.

[12] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models.," in *Inter. Conf. on Parallel Processing*, August 1991.

[13] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.," in *Inter. Symp. on Computer Architecture*, June 1990.

[14] C. Gniady and B. Falsafi, "Speculative Sequential Consistency with Little Custom Storage," in *Inter. Conf. on Parallel Architectures and Compilation Techniques*, September 2002.

[15] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?," in *Inter. Symp. on Computer Architecture*, May 1999.

[16] A. Gupta, W. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," in *Inter. Conference on Parallel Processing*, August 1990.

[17] L. Hammond *et al*., "Transactional Memory Coherence and Consistency," in *Inter. Symp. on Computer Architecture*, June 2004.

[18] M. D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer*, August 1998.

[19] M. Kirman, N. Kirman, and J. F. Martinez, "Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors," in *Inter. Symp. on Microarchitecture*, 2005.

[20] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez, "Checkpointed Early Load Retirement," in *Inter. Symp. on High Performance Computer Architecture*, February 2005.

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

[22] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," *IEEE Computer*, March 1992.

[23] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *Inter. Symp. on Microarchitecture*, November 2002.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Inter. Symp. on High Performance Computer Architecture*, February 2003.

[25] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton, "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors," in *Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[26] M. S. Papamarcos and J. H. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *Inter. Symp. on Computer Architecture*, June 1984.

[27] 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 *Symp. on Parallel Algorithms and Architectures*, June 1997.

[28] 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.

[29] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual Flow Pipelines," in *Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[30] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for Store-wait-free Multiprocessors," in *Inter. Symp. on Computer Architecture*, June 2007.

[31] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, April 1996.