BulkCommit: Scalable and Fast Commit of Atomic Blocks in a Lazy Multiprocessor Environment *

Xuehai Qian†, Josep Torrellas
University of Illinois, USA
{xqian2, torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

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

Depei Qian
Beihang University, China
depeiq@buaa.edu.cn

ABSTRACT

To help improve the programmability and performance of shared-memory multiprocessors, there are proposals of architectures that continuously execute atomic blocks of instructions — also called *Chunks*. To be competitive, these architectures must support chunk operations very efficiently. In particular, in a large manycore with lazy conflict detection, they must support efficient chunk commit.

This paper addresses the challenge of providing scalable and fast chunk commit for a large manycore in a lazy environment. To understand the problem, we first present a model of chunk commit in a distributed directory protocol. Then, to attain scalable and fast commit, we propose two general techniques: (1) Serialization of the write sets of output-dependent chunks to avoid squashes and (2) Full parallelization of directory module ownership by the committing chunks. Our simulation results with 64-threaded codes show that our combined scheme, called *BulkCommit*, eliminates most of the squash and commit stall times, speeding-up the codes by an average of 40% and 18% compared to previously-proposed schemes.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - Parallel Processors.

General Terms

Design.

Keywords

Shared-Memory Multiprocessors, Cache Coherence, Bulk Operation, Atomic Blocks, Hardware Transactions.

1. INTRODUCTION

A class of recently-proposed shared-memory architectures attempts to improve both performance and programmability by continuously executing blocks of consecutive dynamic instructions from the program in an atomic manner. These blocks are also called *Chunks*. These architectures broadly include TCC [8, 12], Bulk [7, 17, 24], Implicit Transactions [25], ASO [26], InvisiFence [3], DMP [10], and SRC [16] among others. This execution mode has been shown to be useful to support transactional memory [8, 12, 16] and high-performance strong memory consistency [3, 7, 24, 26]. It can also be used to provide a platform for novel, high-performance compiler transformations (e.g., [1, 5, 15]). Finally, this model has also been proposed for parallel program development, such as for deterministic execution [10] and parallel program replay [14].

For these architectures to be competitive, they must support chunk operations very efficiently. For this reason, researchers have analyzed many variations in the design of these architectures. In particular, one decision is whether to use lazy or eager detection of conflicts between chunks. In the former, chunks execute obliviously of each other and only check for conflicts at the time of commit; in the latter, conflict checks are performed as the chunk executes. There are well-known pros and cons of each approach, which have been discussed in the related area of transactional memory (e.g., [4, 23]).

In this paper, we focus on an environment with lazy conflict detection. In this environment, when we consider a large manycore with directory-based coherence, a major bottleneck is the chunk commit operation. The reason is that a chunk must appear to commit all of its writes atomically — even though the addresses written by the chunk map to different, distributed directory modules. In addition, a commit may have to compete against other committing chunks that have accessed some of the same addresses — hence prohibiting concurrent commits.

There are several proposals that specifically target this commit bottleneck, namely Scalable TCC [8], SRC [16], and Scalable-Bulk [17]. While these designs have improved the distributed commit operation substantially, commit still remains a major source of overhead, especially when dealing with high processor counts, committing chunks that have accessed data mapped to many directories, and small chunk sizes.

This paper focuses on the challenge of providing scalable and fast chunk commit for a large manycore in a lazy environment. It makes three contributions:

- To understand the problem, it presents a new model of chunk commit in a distributed directory-based protocol, and shows how past schemes map to it.
- It introduces two new, general techniques to attain scalable and fast commit. The first one, called *IntelliSquash*, is the serialization of the write sets of output-dependent chunks. This technique eliminates chunk squashes due to WAW conflicts.
- The second technique, called *IntelliCommit*, is the full parallelization of how committing chunks attain ownership of directory modules. This technique speeds-up the critical path of the commit operation.

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

†Xuehai Qian is now with the University of California, Berkeley. His email address is xuehaiq@eecs.berkeley.edu.

MICRO'46, December 7–11, 2013, Davis, CA, USA.

The combined scheme, called *BulkCommit*, is evaluated with simulations of 64 processors running PARSEC and SPLASH-2 codes. BulkCommit eliminates most of the squash and commit stall times in the execution. The resulting codes run an average of 40% and 18% faster than on previously-proposed schemes.

In this paper, Section 2 motivates the problem; Section 3 explains the model; Sections 4 & 5 present IntelliSquash and IntelliCommit; Section 6 examines design complexity; Section 7 evaluates the design; and Section 8 discusses related work.

2. MOTIVATION

2.1 Difficulties in Chunk Commit

In a chunk-based distributed-directory protocol with lazy conflict detection, the chunk commit operation is critical. To see why, consider the execution preceding the commit. As the chunk executes, cache misses bring lines into the cache, but no written data is made visible outside of the cache. At commit, the writes must be made visible to the rest of the system. This does not mean writing data back to memory, but requires updating the coherence states of the distributed caches and directories — and squashing dependent chunks. Each commit needs to appear atomic, and all the chunks of all the processors need to appear to commit in a total order.

There are three existing proposals that target the commit bottleneck in this lazy, distributed environment: Scalable TCC [8], SRC [16], and ScalableBulk [17]. We will describe their operation in Section 3.3, after we present our commit model. These schemes attempt to overlap, as much as possible, the commit of chunks that have not accessed common addresses (i.e., non-conflicting chunks). At the same time, they must detect when the chunks are conflicting and serialize their commit.

The overall commit overhead in a program mainly depends on three parameters: the number of processors, the average number of directories that map data accessed by a chunk, and the chunk size. As the number of processors increases, the relative overhead of commit increases. This is because, while different chunks can execute in parallel, they have to (appear to) commit serially. If a chunk accesses data mapped to many directories, its commit needs to involve and coordinate many directories, adding overhead. Finally, if the chunk is small, its execution accounts for relatively less time, and the commit for relatively more.

2.2 The Commit Overhead Matters

The three existing schemes are evaluated in three papers: Scalable TCC with simulations of SPEC, SPLASH-2, and other codes for 64 processors [8]; SRC with simulations of synthetic traces for 64-256 processors [16], and ScalableBulk with simulations of SPLASH-2 and PARSEC for 64 processors [17]. The data in each paper largely suggests that, after the scheme is applied, chunk commit is overlapped with chunk execution and, hence, does not affect a program's execution time.

However, the scenarios described in these papers are different from the one we focus on: their chunks are large, their chunks access data mapped to few directories, and their processors have hardware to support multiple outstanding chunks. Each of these effects hides the overhead of commit.

Specifically, Scalable TCC uses large transactions that often have 10K-40K instructions. Such chunks are obtained by instrumenting the application and positioning the transactions in the best places in the code. In SRC, the authors consider synthetic models of programs with chunks larger than 4K instructions.

In this paper, we focus on a chunked environment where the code is *executed as is*, without program instrumentation or analysis of where it is best to create chunk boundaries. Hence, chunks are smaller to minimize inter-thread dependences. Also, unpredictable cache overflows and system calls further reduce the chunk size. We expect average chunks of about 2,000 instructions.

In addition, in Scalable TCC, each chunk commit accesses very few directories: in all but two codes, 90% of the commits access only one or at most two directories. In SRC, the synthetic model is based on the Scalable TCC data.

In this paper, we focus on an environment where we cannot fine-tune the code that goes into a chunk. A chunk accesses data from multiple localities. Hence, we observe that a typical chunk accesses data mapped to several directories.

Finally, in ScalableBulk, the architecture evaluated is such that each processor has hardware to support two outstanding (or *active*) chunks (Table 2 in [17]). This means that two chunks can be executing at the same time. This approach hides the overhead of commit: while one chunk is committing, the next one is executing. However, this approach has hardware and control cost: twice as many checkpoints and signatures, rollback of partial state (one of the two chunks), and the need to handle the data access overlap of the two chunks with stalling or data forwarding, as we will see.

In this paper, we focus on an inexpensive system with a single chunk per processor. As a result, when a processor is committing a chunk, it has to wait until it is sure that the chunk will successfully commit before executing the next chunk. This issue exposes commit overhead.

In our discussion, we assume a tiled manycore architecture with directory coherence. Each tile has a processor, a private L1 cache, and a portion of the shared L2 cache and directory.

3. LIFETIME OF A CHUNK: A MODEL

3.1 The Model

The lifetime of an chunk in a directory-based lazy environment has three sequential stages: *Execution*, *Grouping*, and *Propagation* (Figure 1). The last two, combined, form the *Commit* stage.

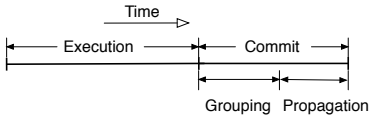

Figure 1: Model of the lifetime of a chunk.

Execution is the time while the processor executes the chunk instructions. Reads and writes bring lines into the cache, but no written line is made visible to other processors, which continue to read the non-speculative value of the line from memory or other caches. Execution ends when the last instruction of the chunk completes.

Grouping is the first stage of the commit. It sets the relative order of any two (or more) chunk commits that cannot proceed concurrently. The Grouping stage of a chunk attempts to establish a coordination between all the directory modules that map the lines accessed by the chunk. We call this process *grabbing* the directories. Two chunks that are not allowed to commit concurrently because they have accessed common memory lines will conflict when they try to grab the same directory. Only one of the chunks will succeed and proceed to the Propagation stage; the other will stall until the first one has completed its Propagation. As soon as the Grouping stage completes successfully, the commit is guaranteed to succeed (i.e., the chunk will not be squashed). Hence, at this point, the processor can start to execute the next chunk of the program.

Propagation involves making the stores in the chunk visible to the rest of the system. While Propagation appears atomic, it takes a

certain time, as it requires updating directory entries and sending invalidations to other caches in the machine. Propagation may involve no data transfer. While Propagation takes place, the directory modules prevent accesses by other processors to the directory entries of the lines accessed by the committing chunk; this ensures the atomicity of the Propagation. After Propagation completes, no processor can see the old values of the updated lines. Propagation can overlap with the execution of the next chunk in the initiating processor.

A chunk in the Propagation stage squashes other processors' chunks in the Execution or Grouping stages if the latter chunks are data dependent on the former chunk. In current systems, this is implemented by comparing the Propagating chunk's write set (i.e., the addresses written) to the read and to the write sets of the other chunks. If there is an overlap in a chunk, that other chunk is squashed — ostensibly because it has accessed stale data.

3.2 Stalling in the Grouping Stage

Two chunks are said to be conflicting if they have overlapping footprints. This means that some addresses in the write set of one chunk are also present in the read or in the write sets of the other. When two conflicting chunks perform Grouping concurrently, only one succeeds, and the other stalls in the middle of its Grouping until the first one completes the Propagation.

If the stalled chunk (call it C_1) only has WAR conflicts with the successful one (call it C_0), C_1 will not be squashed. Hence, C_1 could be allowed to proceed with its Grouping as soon as C_0 finishes Grouping, without waiting for the end of C_0's Propagate. While this is possible, we discourage it because overlapping Propagates may result in processors receiving messages out-of-order, which complicates the protocol.

3.3 Applying the Model to Existing Protocols

We apply our model to different protocols in the literature.

Scalable TCC [8]. In the Grouping stage, a chunk first obtains a global-order transaction ID (TID) from a central agent. This operation logically serializes competing commits. In addition, the committing processor sends messages to *all* of the directory modules in the machine, to identify when the Grouping stage is over and it can start Propagation. These messages are a Skip message to the directories not in the chunk write set, and (potentially several) Probe messages to the directories in the chunk's write and read set. When all of the directories have completed the conflicting previous commits, then the probes succeed. Then, the Propagation stage starts. It involves sending the chunk's write addresses to the directories in the write set. In this protocol, two chunks can commit concurrently only if they use *different directory modules*. If any directory in the write set of one chunk appears in the read or write set of the other chunk, the commits are serialized — even if the addresses accessed by the two chunks are disjoint.

SRC [16]. SRC optimizes the Grouping stage by eliminating the centralized agent and broadcasting step. A processor with a committing chunk tries to grab the directory modules in the chunk's access set in a *sequential* manner, from the lowest-numbered one to highest-numbered one. Two committing chunks that try to grab the same directory get ordered. The one that fails waits until the successful one fully completes its commit. As in Scalable TCC, two chunks that use the same directory module but access non overlapping addresses cannot commit concurrently. Once a committing chunk grabs all of its directories, it proceeds to the Propagation stage. The Propagation stage executes as in Scalable TCC. In the paper, the authors also outline an optimization called SEQ-TS that improves on SRC. However, there are few details on how it works.

ScalableBulk [17]. ScalableBulk further optimizes the Grouping in two ways. First, although the directories are still grabbed in a *sequential* manner, the transaction does not involve repeated round trips from the initiating processor to the directory modules — the directory modules organize themselves as a group. Secondly, two chunks can concurrently grab the same directory module as long as the addresses they accessed do not overlap. This is accomplished by using address signatures to represent access sets, and intersecting them to detect overlap. In the Propagation stage, ScalableBulk does not propagate addresses; only signatures are passed between nodes, which is cheaper.

3.4 Sources of Inefficiency

As we have seen, successive proposals have progressively optimized the commit — especially the Grouping stage, which is the one in the critical path. However, there are still major bottlenecks. An obvious one is that the Grouping stage still requires grabbing directory modules in a *sequential* manner.

A second, subtler one, is that a chunk in the Propagation stage squashes a second chunk in the Execution or Grouping stages even if there are only write-after-write (WAW) conflicts between the chunks. These are output dependences rather than true dependences, and should not cause a squash.

In this paper, we eliminate these bottlenecks with two new designs, called *IntelliCommit* and *IntelliSquash*. With them, we make chunk commit highly efficient, and attain high performance in executions with high processor counts and chunks that are small and access several directories. Supporting small chunks is important because, in some workloads, large chunks are not an option: large chunks suffer frequent dependences that lead to squashes. In the following, we present the two new techniques.

4. IntelliSquash: NO SQUASH ON WAW

4.1 Basic Idea

In existing chunk-based protocols, if a chunk currently in the Propagation stage wrote to an address that a later chunk C has read, C will get squashed. This is needed because C has read stale data. However, most protocols also squash C even if it has *only written* to the address that the Propagating chunk wrote. This is a WAW conflict and should not induce a squash.

Past work on Thread-Level Speculation (TLS) (e.g., [11, 13, 21, 22]) has, in some cases, provided support to avoid squashes on WAW conflicts. However, TLS requires that this is done in the context of a total order of tasks, which introduces other unnecessary complications. In the context of transactional systems, where there is no pre-determined task order, there have been a few proposals that avoid squashes on WAW conflicts. They are DATM [19], SONTM [2], and BulkSMT [18], which dynamically *forward* data between concurrently-executing, dependent chunks, and then force an order to their commits. Such protocols, however, add an extra layer of complexity that we want to avoid. Our goal is to augment a baseline chunk protocol with a *general primitive* that prevents squashing the chunk if there are only WAW conflicts — without having to record the dependences, forward data, or order the chunks. We call our technique *IntelliSquash*.

To understand how IntelliSquash works, consider a conventional multiprocessor where the write buffers of two processors (P_0 and P_1) each have a store (w_0 and w_1) to the same line. Suppose that the line is in state Shared in the caches of both processors. The stores will drain and get ordered in some way without requiring squash and re-execution. Specifically, both stores issue requests for line ownership. The one that reaches the line's home directory first (say

w_0) gets serialized before the other, and invalidates the line from the other cache (P_1's). The line becomes owned by P_0 and w_0 is applied. Later, w_1 misses in its cache, reaches P_0's cache, obtains the line (and invalidates it), and brings it to P_1's cache, where w_1 is applied. The effects of the two stores are serialized without store re-execution.

Current chunk commits do not work like this. In a typical implementation, the L1 cache serves the same purpose for a chunk as the write buffer for an individual write. The difference is that, in a conflict, the data in the write buffer is not affected by an external invalidation, while the cache lines that the chunk updated get invalidated. Hence, the chunk's updates are lost. This data can only be recovered by squashing and re-executing the chunk.

IntelliSquash uses the idea of the write buffer to serialize, without any squash, the commits of two chunks that only have WAW conflicts. Specifically, consider the case when the only conflict between two chunks (i.e., C_0 and a later one C_1) is that C_1 had written (and not read) a variable that C_0 currently in the Propagation stage has written. IntelliSquash invalidates the corresponding line from C_1's cache, but the write of C_1 is not lost. Later, when C_1 commits, its write will be merged with the memory system state. Overall, the writes of the two chunks are applied in a serial manner, without squashes.

4.2 IntelliSquash Design

To support IntelliSquash, we extend the cache with some bits that trigger certain transactions. In the following, we explain the design. Without losing generality, we explain it as extensions to a protocol with signatures.

4.2.1 Additional Cache Bits.

The cache must have the ability to retain speculatively-written data even while it invalidates the *rest of the line*. For this reason, we extend each cache line with one *Absent (A)* bit, and with fine-grain dirty bits (one bit per word or per byte, depending on the granularity that we want to support). Figure 2 shows the design. In Figure 2(a), we show a conventional 4-word cache line with a Speculative (Sp), Valid (V) and Dirty (D) bit. In Figure 2(b), we show the line under IntelliSquash: it adds the Absent (A) bit and per-word Dirty (d) bits (since we assume word granularity).

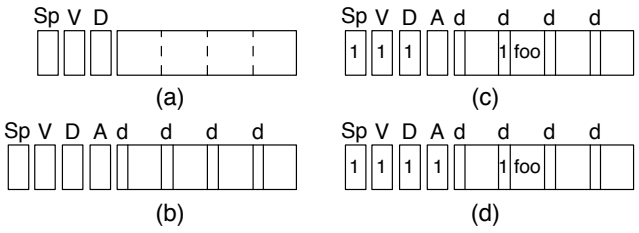

Figure 2: Cache line augmented for IntelliSquash.

To see how these bits work, assume that chunk C_1 writes *foo* to the second word of a line and misses in its cache. As in conventional schemes, the line is brought from the memory, the word is updated, and *Sp*, *V* and *D* are set. In addition, the second word's *d* bit is set (Figure 2(c)). Suppose that, at this point, the local processor receives the write set (e.g., in a signature) from a remote chunk C_0 that is in the Propagation stage. Further, the write set overlaps with C_1's write set (at a line granularity) but *not* with C_1's read set.

In this case, IntelliSquash sets the line's *A* bit, transitioning it to the *Dirty Absent* state — effectively invalidating the words with *d*=0. As seen from the directory, this state is the same as Invalid (the processor is not a sharer of the line anymore in the directory). However, as C_1 continues to execute, accesses to words with *d*=1 are satisfied as cache hits. An access to a word with *d*=0 induces a cache miss that brings-in the line (but only overwrites the part of the line with *d*=0), marks the processor as sharer in the directory, and clears the Absent bit. This transaction is called a *Merge* transaction. Finally, the commit of C_1 will involve issuing Merge transactions for all its Dirty Absent lines.

From this discussion, we see that IntelliSquash only allows a line in a processor's cache to transition to Dirty Absent if the processor has not read the line. Strictly speaking, if the read had occurred after a write to the same word, it would be correct to also transition to Dirty Absent. However, we use the above algorithm for simplicity.

4.2.2 Merge Transaction on a Miss.

When a chunk accesses a word with *d*=0 in a line in Dirty Absent state, the cache initiates a Merge transaction. The transaction proceeds like a miss, obtaining the current non-speculative version of the line, and recording the requesting processor as a sharer in the directory. Once the line arrives, it only updates the words that had *d*=0 and clears the *A* bit, transitioning to the Dirty state locally. The *Sp*, *V* and *D* bits remain set. If the access was a store, the *d* bit of the word is set.

The lazy protocol that we assume is like BulkSC [7], in that a write miss to a line by a chunk appears as a read miss outside the cache: the state of the line in the directory is set to Shared, although the line is (speculatively) Dirty in the cache.

4.2.3 Merge Transactions on a Commit.

When a chunk commits, the commit involves generating Merge transactions for all its lines in Dirty Absent state. These transactions obtain individual lines from main memory (if the directory has them as Shared) or from another cache (if the directory has them as Dirty in a cache). As usual, all of the lines in the write set of the committing chunk (including those in Dirty Absent state) are invalidated from the other caches, and are marked by the directory as Dirty in the committing processor.

To see how this is done with signatures, consider a protocol like BulkSC. In the Propagation stage, the committing chunk's write signature (W) is expanded in the directory into its constituting addresses. For each such address, the line is marked as Dirty in the directory (and owned by the committing processor). In addition, W is sent to the sharer processors to invalidate the cache lines whose addresses are in W. In IntelliSquash, it is during this time that the Dirty Absent lines are sent to the committing processor. The directory recognizes such lines because, although they belong to W, they are marked in the directory as not present in the committing processor. Recall also that a Dirty Absent line may be (non-speculative) Dirty in another cache, in which case the directory first asks the owner to write it back (and invalidate itself) before forwarding the line to the committing processor. At the end of the commit, all the lines in the write set of the committing chunk are marked in the directory as Dirty in the committing processor.

4.2.4 Effects of False Positives.

When W expands in the directory, it may generate line addresses that were not in the committing chunk's write set. They are false positives. Such false positives can at most create unnecessary traffic, but never affect correctness. To see why, consider a line address emerging from the expansion. Its directory entry can be in one of the four possible states in Table 1 — depending on whether the Dirty bit (D) and/or the presence bit of the committing processor (K) in the Bit Vector are set. The table assumes an MSI/MESI protocol; other protocols may require some changes. The table shows

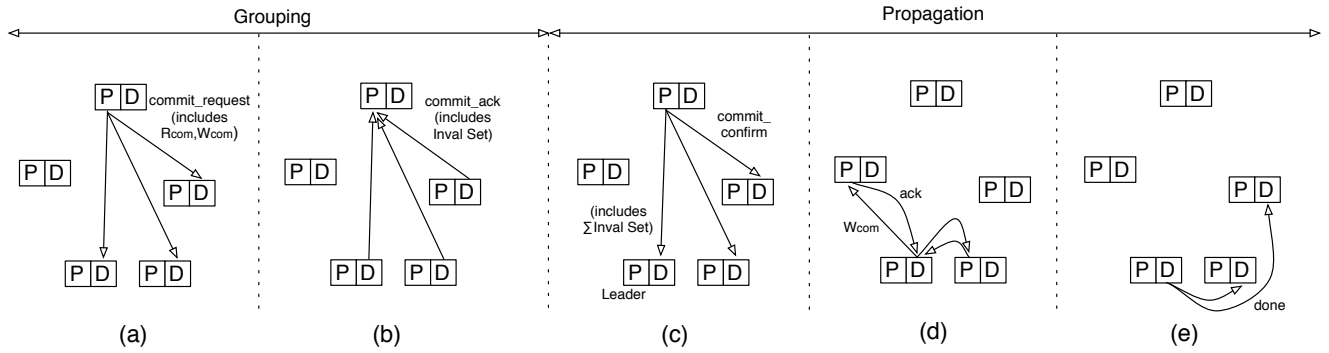

Figure 3: IntelliCommit commit protocol. The figure shows five nodes with processor (P) and directory (D).

the actions that IntelliSquash takes. Note that, in each case, it may be a line not part of the committing chunk's write set.

Current Entry State		Action in IntelliSquash	Action in BulkSC
Dirty Bit	Committing Proc Bit in Bit Vector		
D=0	K=1	Invalidate sharer caches Clear rest of bit vector Set Dirty bit	Same as IntelliSquash
D=0	K=0	Merging transaction: Provide line to committing proc Invalidate sharer caches Clear bit vector and set the bit for the committing processor Set Dirty bit	Do nothing: false positive
D=1	K=0	Merging transaction with owner: Owner writes back & invalidates Provide line to committing proc Clear bit vector and set the bit for the committing processor	Do nothing: false positive
D=1	K=1	Do nothing: false positive	Same as IntelliSquash

Table 1: Directory entry states after signature expansion.

If the directory has D=0 and K=1, we assume it is a normal write. IntelliSquash invalidates the other sharer caches, clears the rest of the Bit Vector (only its bit remains), and sets D. These are the same actions as in BulkSC. If it is a false positive, we are incorrectly changing the directory to point to the processor. This is incorrect but easy to recover from gracefully [7]: it is like a processor reading a line in Exclusive state and then evicting it from its cache silently. On a future request, the protocol will realize that the cache does not have the line and will provide it from the memory.

If the directory has D=0 and K=0, IntelliSquash initiates a Merging transaction. It involves the same operations as in the previous case plus providing the line to the committing processor and setting its K bit in the directory to 1.

If the directory has D=1 and K=0, IntelliSquash initiates a Merging transaction with owner. The directory requests the line from the owner processor and sends the line to the committing one. The old owner invalidates its cache entry. The K bit of the old owner is cleared, while the one for the committing processor is set. The Dirty bit stays set.

The two cases just described are false positives in BulkSC and no action is taken. In IntelliSquash, if they are false positives, we have simply forwarded an unrequested line to the committing processor and marked it as the owner. The cache in the committing processor may choose to take-in the line or silently reject it. In the latter case, we have a situation like in the first case above.

The final case, where the directory has D=1 and K=1, is a false positive. Hence, IntelliSquash (like BulkSC) does nothing. Overall, we see that IntelliSquash always works correctly.

5. IntelliCommit: PARALLEL GRABBING

5.1 Basic Idea

While the critical path of the commit operation in a lazy environment is the Grouping stage (Section 3.1), the recently-proposed designs still execute it inefficiently. Specifically, the Grouping stage for a chunk in SRC [16] and ScalableBulk [17] grabs the directory modules in the access set of the chunk in a *sequential* manner. Hence, if a chunk commit needs to grab many directory modules, Grouping is slow.

To speed-up Grouping, this section proposes *IntelliCommit*, the first design where the Grouping operation grabs the directory modules in parallel. The idea is for the committing processor to send *commit_request* protocol messages to all of the relevant directory modules, get their responses directly, and finally send them a *commit_confirm* message. The main challenge is to correctly arbitrate multiple concurrent chunk commits. In the following, we present the commit protocol and its arbitration mechanism, and discuss its correctness.

5.2 IntelliCommit Commit Protocol

Our IntelliCommit protocol uses address signatures like ScalableBulk [17] to encode the read and write sets of a chunk. Figure 3 shows the protocol, using a scalable architecture where each node has a processor and a directory module. During the execution of a chunk, the processor records the set of directory modules that map the lines accessed in the chunk. Then, when the chunk finishes execution, IntelliCommit sends a *commit_request* message with the chunk's signatures (R_{com} and W_{com}) to such directories. Figure 3(a) shows the case with three such directories. Then, these directories respond with *commit_ack* messages (Figure 3(b)), which include *Invalidation Sets* (i.e., the set of nodes that should receive invalidations, according to the local directory's sharing information). In addition, the directories use the R_{com} and W_{com} signatures to disallow subsequent accesses in the directories to the lines in R_{com} and W_{com}, preventing a conflicting chunk from Grouping concurrently.

When the processor receives all of the expected *commit_acks*, a group has been formed, and the Grouping stage ends. Then, the Propagation stage starts, while the processor proceeds to execute the next chunk. Propagation starts with a *commit_confirm* message from the processor to the same directories (Figure 3(c)). As in ScalableBulk, there is a default policy that designates a *Leader* in each group — e.g., the lowest-numbered node. As part of the *commit_confirm* message to the Leader, the committing processor includes the union of all of the Invalidation Sets.

The Leader then sends W_{com} to all of the nodes in the combined Invalidation Set, for disambiguation (and possible chunk squash) and cache invalidation. Figure 3(d) shows the case with two such nodes. When all the nodes in the set indicate with an acknowledgment that such operation is completed, the Leader sends a *done* message to the other directory modules in the group, so that they make the locked directory entries accessible again (Figure 3(e)). The commit is done.

Between the time that a processor sends the *commit_requests* for a chunk (Figure 3(a)) and the time it receives all the *commit_acks* (Figure 3(b)), the processor may receive a *W* signature from the commit of another chunk that squashes the current chunk. In this case, the processor immediately sends a *commit_cancel* message to all the directories it had sent the *commit_requests*. In addition, it will discard any incoming *commit_acks*. The directories, on reception of the *commit_cancel*, discard the signatures of the chunk, make its directory entries accessible, and consider the commit aborted.

A processor cannot receive a *W* signature that squashes its chunk C_1 after it has already received all the *commit_acks* for C_1 (Figure 3(b)). The reason is that the directories cannot form the group for C_1 (by all sending the *commit_ack* to the processor), if another chunk C_0, which has signatures overlapping with those of C_1, is currently in the Propagation stage and, therefore, can squash C_1. If such a chunk C_0 existed, it would have prevented the directories where C_0 and C_1 overlap from sending a *commit_ack* to the processor of C_1 in the first place. Hence, C_1 cannot have formed its group. To see this, we now describe how chunks compete in the Grouping stage.

5.3 States of a Committing Chunk in a Directory Module

From a directory module's viewpoint, a chunk commit starts when the directory receives a *commit_request*, and it ends when the directory receives acks from all of the invalidated caches (if it is the leader) or the *done* message from the leader (otherwise). It may also end early if the directory receives a *commit_cancel* message. During the commit of the chunk, the chunk goes through the states of Figure 4 in a directory module.

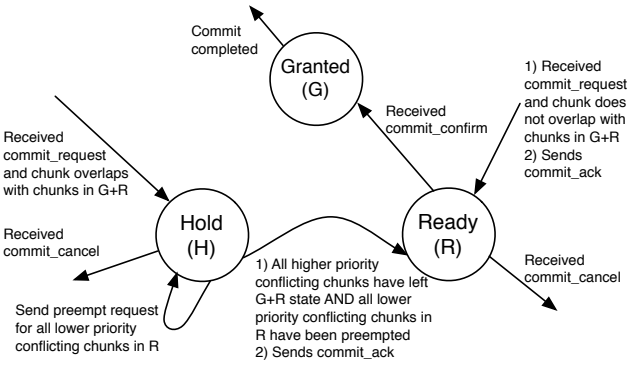

Figure 4: State transitions of a committing chunk in a directory module.

Assume that a directory module receives a *commit_request* for a chunk C_0 (which includes its signatures). If there is no other chunk in the directory module, the directory responds with a *commit_ack* and sets the chunk state to Ready (R). Later, when the directory receives the *commit_confirm*, the commit is irreversible, and the chunk transitions to Granted (G). It will exit G when the commit completes.

The same process also takes place if the arriving chunk's signatures do not overlap with any of the signatures of the chunks C_i that are currently in R or G states — specifically, $R_0 \cap W_i$, $R_i \cap W_0$, and $W_0 \cap W_i$ are null.

However, if the signatures overlap, the arriving chunk C_0 is set to the Hold (H) state, and the directory does not send *commit_ack*. Each chunk in H has a *Wait Set* list, with the chunks currently in G or R that it overlaps with. Moreover, C_0 compares its priority to that of its overlapping chunks in R. If C_0's priority is higher, it attempts to preempt them, by sending preempt requests. We will see in Section 5.4 how we assign priorities and how preemption works.

C_0 moves from H to R, and the directory sends a *commit_ack* as soon as (i) all of its higher-priority conflicting chunks have left the G+R states and (ii) all of its lower-priority conflicting chunks in R have been preempted.

At all times, the signatures of the chunks in G, R, or H in the directory module are buffered. When the directory module receives a *commit_cancel* for a chunk in H or R, the commit of that chunk terminates.

5.4 ChunkSort: Distributed Algorithm to Order Chunks

We now describe how we order conflicting chunks.

5.4.1 Significance.

In a directory protocol for lazy-conflict chunks, it is challenging to devise a low-overhead, scalable Grouping stage. The difficulty comes from having to handle concurrent requests from potentially conflicting chunks in a fully distributed manner. As an example, consider two committing chunks that need to grab two directory modules each. If the two chunks either (i) use different directory modules or (ii) use the same directory modules but their signatures do not overlap, then the Grouping stages of the two chunks can proceed in parallel. Otherwise, one of the chunks will succeed in building a group, while the other will stall.

The decision of which chunk succeeds must be the same in all of the common directories. Unfortunately, messages may arrive with different timings at different directories and, initially, different directories may make opposite decisions. To reach a single decision, the protocol needs two supports. First, there has to be a known, common policy that, on a conflict, mandates which chunk has priority. Secondly, a directory module has to be able to change its decision if, due to reasons of message timing, the directory initially took an inconsistent decision. This is the *preemption* operation mentioned above. Consequently, preemption is not a performance issue, but a correctness one.

In this section, we describe the priority policy and the preemption algorithm, called *ChunkSort*.

5.4.2 Priority Policy.

IntelliCommit has a priority policy that directories use *locally* when two chunks with overlapping signatures want to grab the same directory module. The main requirement is fairness. In our design, each processor has a Linear Feedback Shift Register (LFSR), which generates a random number to be included in the *commit_request*. When two requests conflict, the directory gives priority to the one with the lower random number. In a tie, the lower ID wins.

5.4.3 Preemption Algorithm: ChunkSort.

ChunkSort's approach is to make decisions based on information available locally in the directory module, and only introduce a small number of inter-node messages to accomplish the preemp-

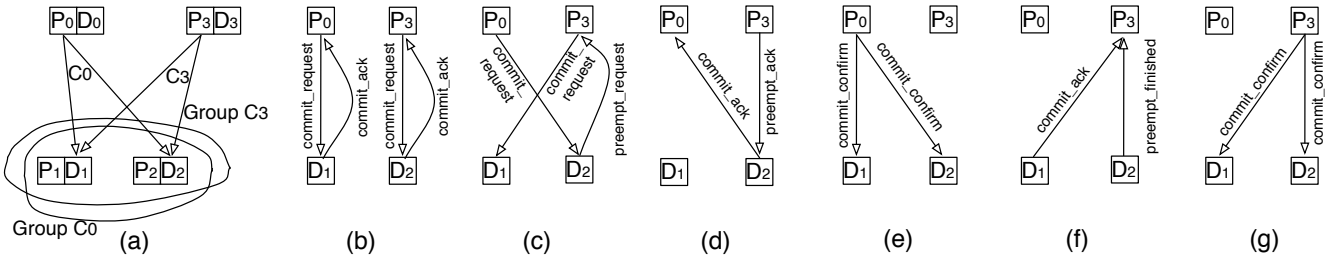

Figure 5: Example of a chunk preempting another.

tion. The ChunkSort algorithm is run by the controller in a directory module when there is a chunk in state H that could be in state R except for a conflicting, lower priority chunk already in R. In this case, correctness requires that preemption be attempted, since it is possible that, in another directory module that both chunks also need to grab to commit, the higher and lower priority chunks are in the opposite states due to message arrival times. Hence, not performing preemption could result in deadlock.

Let us call C_{low} the low-priority chunk in state R, P_{low} the processor where C_{low} was running, C_{high} the high-priority chunk in state H, and P_{high} the processor where C_{high} was running. Preemption starts with the directory module sending a *preempt_request* message to P_{low} on behalf of C_{high}. If P_{low} has already started its Propagation stage (because it has already received all its *commit_acks*), then P_{low} simply responds with a *preempt_nack*. It is too late to perform a preemption, and C_{high} has to wait until the commit of C_{low} completes — unless it is squashed before.

Otherwise, the preemption occurs. P_{low} responds with a *preempt_ack* and records which directory requested the preemption and for which chunk. P_{low} will not proceed to Propagation yet; it may keep receiving *commit_acks* for C_{low}, but will take no action. When the directory receives the *preempt_ack*, it moves C_{high} to state R, and places it ahead of C_{low}. It also sends a *commit_ack* to P_{high} for C_{high}. This same process may occur in several directories.

It is guaranteed that C_{high} will eventually enter the R state in all of its directories, as it will preempt C_{low} everywhere. Thus, the Grouping stage of C_{high} will be able to complete. Once C_{high} completes the full commit, the directories that preempted C_{low} send a *preempt_finished* message to P_{low}.

P_{low} has to receive *preempt_finished* messages from all of the directories that initially sent *preempt_requests* and were granted. Once P_{low} has received all of the *preempt_finished* and also all its *commit_acks*, then P_{low} proceeds to Propagate, by sending *commit_confirm*.

5.4.4 State for ChunkSort: Preemption Vector (PV).

To support preemptions, a processor needs to record which other chunk(s) preempted its chunk, and know when they complete their commit. We support this with a *Preemption Vector (PV)* in the processor. In a machine with N processors, the PV has N-1 counters (one for each of the other processors). Each counter can count up to N-1.

Suppose that chunk C running in processor P is committing and it gets preempted by chunk C_j running in processor P_j. In this case, the PV[j] of processor P will count the number of *preempt_request* messages that P has received and granted for C_j. Later, as processor P receives *preempt_finished* messages for the chunk committed by P_j, it decrements PV[j]. Note that a chunk may be preempted by multiple chunks in multiple directories and, therefore, multiple entries in PV may be non-zero. Hence, only when a processor's PV reaches zero for *all of its entries* can the processor restart the commit of its preempted chunk.

In addition, a processor also needs to record the number of *commit_acks* that it has received for the chunk that it tries to commit. As it restarts the commit of the preempted chunk, it can only send the *commit_confirm* when it has received all of the *commit_acks*.

A preempted chunk may be squashed due to a dependence. In this case, the processor sends the usual *commit_cancel* and clears its PV and the count of the number of *commit_acks* received.

5.4.5 Preempting Multiple Chunks.

It is possible that a chunk in state H in a directory needs to preempt multiple chunks that are in state R in the directory. In this case, ChunkSort works seamlessly. The directory sends *preempt_requests* to multiple processors. Similarly, a chunk in state R in two directories may be preempted by a different chunk in each directory. Here, ChunkSort also works seamlessly. The processor executing the chunk receives *preempt_requests* from the two directories and updates two of its PV entries.

5.5 Example

To illustrate IntelliCommit and its relation to IntelliSquash (Section 4), we show in Figure 5 an example of a chunk preempting another. As shown in Figure 5(a), processors P_0 and P_3 want to commit chunks C_0 and C_3, respectively, and both need to grab directories D_1 and D_2. We assume that their signatures overlap and that C_0 has higher priority.

Both chunks start the Grouping stage at the same time. Let us assume that a *commit_request* from P_0 arrives at D_1 first, and one from P_3 arrives at D_2 first. The directories place the chunks in R state and respond with *commit_acks* (shown in a simplified format in Figure 5(b)). As the second pair of *commit_requests* arrive at the directories, since the incoming signatures overlap with those in state R, the chunks are placed in H state (C_0 is H in D_2 and C_3 is H in D_1). Since C_0 has a higher priority than C_3, C_0 attempts to preempt C_3 in D_2 by sending a *preempt_request* from D_2 to P_3 (Figure 5(c)).

Since P_3 has not received all of its *commit_acks* yet, it allows the preemption, replying with a *preempt_ack* to D_2. On reception of the message, D_2 moves C_0 to R state and, on behalf of C_0, sends a *commit_ack* to P_0 (Figure 5(d)). At this point, the commit of C_0 enters the Propagation stage and P_0 sends *commit_confirms* to the two directories (Figure 5(e)). In the meantime, C_3 is waiting in both directories, unable to complete its Grouping stage.

During C_0's Propagation stage, C_3's commit can be affected in one of three different, exclusive ways — depending on the type of overlap that exists between C_0 and C_3's signatures.

- **If $W_{C0} \cap R_{C3}$ is not null**, it means that C_3 has true dependences with C_0 and has to get squashed. When P_3 receives the W_{C0} signature for disambiguation, it squashes C_3, terminating its commit. P_3 sends *commit_cancel* to directories D_1 and D_2. When P_3 later receives a *commit_ack* from D_1 or a *preempt_finished* from D_2, it discards them.

Design	Hardware Structures	Control Logic
Base (Inherited from BulkSC and Scalable-Bulk)	– Checkpoint and R/W signatures – Functional units that operate on signatures in cache controller and directory controller	– Processor and cache controller: checkpoint and rollback, address disambiguation using signatures – Directory controller: update directory state using signatures
IntelliSquash	– Per cache line: • A bit • d bit per byte (or word)	– Cache controller: • Merge transaction on a miss: incoming line does not overwrite words with d=1 (Sec. 4.2.2) • Incoming invalidation: set A bit – Directory controller: • Merge transactions on chunk commit: send some lines from memory to the cache to merge (Sec. 4.2.3)
IntelliCommit	– Cache controller: • # of *commit_ack* & *preempt_finished* received • LFSR register to generate random numbers • Preemption Vector: N-1 counters	– Directory controller and cache controller: • Chunk commit protocol (Fig. 3) • State of a committing chunk in a directory module (Fig. 4) • Preemption state machine (Fig. 5)
Removed complexity from ScalableBulk	– Cache controller: Two active chunks per processor • Additional checkpoint and R/W signatures	– Processor and cache controller: Two active chunks per processor • Rollback of partial state (one of the two chunks) • Overlap of two active chunks: stalling or data forwarding – Directory Controller: • ScalableBulk commit protocol (Sec. 3.3): More transient states

Table 2: Estimated design complexity of BulkCommit.

- **Otherwise, if $\mathbf{W}_{C0} \cap \mathbf{W}_{C3}$ is not null**, it means that C_3 has output dependences and no true dependences with C_0. In this case, the IntelliSquash technique of Section 4 prevents the squash of C_3. The lines written by both processors become *Dirty Absent* in P_3's cache, and C_3's updates are not lost. C_3 continues to be stalled in the Grouping stage, with P_3 waiting for a *commit_ack* from D_1 and a *preempt_finished* from D_2.

- **Otherwise, $\mathbf{R}_{C0} \cap \mathbf{W}_{C3}$ is not null**. In this case, C_3 only has WAR dependences with C_0. As per Section 3.2, IntelliCommit stalls the Grouping of C_3 to make the protocol simpler, and C_3 does not get squashed. P_3 simply waits for the same messages as in the previous case.

Once C_0 completes the Propagation stage, its commit is completed. At this point, the directories send messages on behalf of C_3 to P_3: a *commit_ack* from D_1 and a *preempt_finished* from D_2 (Figure 5(f)). If C_3 has not been squashed in the meantime (last two cases above), P_3 transitions to Propagate and sends *commit_confirms* to the two directories (Figure 5(g)).

5.6 IntelliCommit Correctness Properties

In this section, we discuss correctness properties of IntelliCommit.

Atomicity of Commit. If two chunks that attempt to commit concurrently have dependences, their signatures will overlap. Hence, in the common directories, one of the chunks will be forced to stall the Grouping until the other one completes the Propagation. Therefore, the commits are serialized. Signature intersections can have false positives, in which case IntelliCommit stalls (and perhaps squashes) a chunk unnecessarily. However, correctness is not affected. Signature intersections cannot have false negatives.

Consensus of Commit Order for Conflicting Chunks. In the Grouping stage, two conflicting chunks are ordered in the *same way* in all of the common directories. This is guaranteed for two reasons. First, each directory uses the same information, available locally, to order two chunks that are in R or H states. Second, the preemption algorithm guarantees that if a chunk C_0 preempts a second one C_1 in *any* directory module, then such a preemption will also succeed in *all other* common directory modules. This is because as soon as the processor executing C_1 issues a single *preempt_ack*, it will not send the *commit_confirm* until it is informed that C_0's commit has completed (with *preempt_finished* messages).

Liveness. In IntelliCommit, a commit eventually succeeds or fails. To see why, consider the property of consensus of commit order. It ensures that one of the conflicting chunks completes. At this point, if the second chunk has not been squashed, it will resume its Grouping stage. This is because the directories where the first chunk stalled the second one will send *preempt_finish* messages to the second chunk's processor. The processor will respond with a *commit_confirm*.

Deadlock Freedom. The property of consensus of commit order plus the property of liveness ensure that deadlock cannot happen.

Starvation Freedom. If we use a fair policy to decide which of two chunks has priority when they conflict in a directory, there is no starvation. The policy of Section 5.4.2 does not cause starvation.

6. ANALYSIS OF DESIGN COMPLEXITY

Table 2 estimates the design complexity of BulkCommit and also compares it to that of ScalableBulk. We divide complexity into hardware structures and control logic. BulkCommit is composed of a base design from BulkSC [7] and ScalableBulk [17] (Row 1), plus IntelliSquash (Row 2), plus IntelliCommit (Row 3), and minus some components from ScalableBulk (Row 4).

The top three rows are largely self-explanatory. The last row shows two components of ScalableBulk not present in BulkCommit: two active chunks per processor and the ScalableBulk chunk commit protocol. The former requires an additional checkpoint and set of R/W signatures. It also needs control logic to roll back only the state of one of the two chunks, and to handle the overlap of the locations accessed speculatively by the two chunks. Specifically, if the predecessor chunk writes a location and the successor chunk reads it, data forwarding requires some care [7]. In addition, if the predecessor chunk writes to a line and the successor chunk attempts to write to the same line, it has to stall. The successor also has to stall if it attempts to write to a different line of the same cache set. This is because of the Set Restriction [6], which requires that a cache set hold speculative dirty data from only one chunk.

The second component is the ScalableBulk chunk commit protocol, which is more complex than the BulkCommit protocol: it has more transient states. The reason is that it has a higher number of different types of messages received by a directory module before the commit succeeds or fails. In addition, the messages can be received in different orders. To get a *qualitative* insight, we compare the BulkCommit protocol in Figure 3 to the ScalableBulk protocol in Figure 3 of [17]. In the latter, the directory modules coordinate

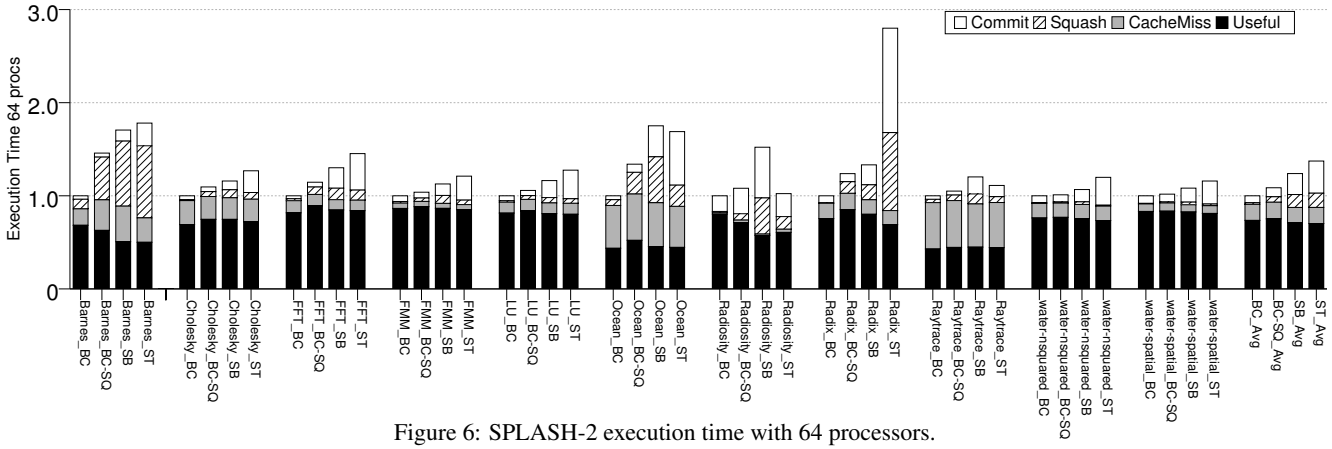

Figure 6: SPLASH-2 execution time with 64 processors.

among themselves to form a group and to resolve collisions. As glimpsed from the figure and shown in Table 5 of [17], the messages exchanged are of different types and can come in a variety of orders, creating transient states. The BulkCommit protocol is simpler.

7. EVALUATION

7.1 Evaluation Setup

We evaluate BulkCommit using simulations of a 64-core multicore using a simulator of processors, caches, and network [9]. The cores issue and commit two instructions per cycle. Memory accesses can be overlapped with instruction execution through the use of a reorder buffer. Each core has a private L1 cache and one bank of the shared L2 cache. Caches are kept coherent using a directory-based scheme that implements the BulkCommit protocol. The cores are connected using an on-chip 2D mesh. A simple first-touch policy is used to map virtual pages to physical pages in the directory modules. Table 3 shows more details. We evaluate the protocols listed in Table 4: BulkCommit, BulkCommit minus IntelliSquash, ScalableBulk, and Scalable TCC. To make the comparison fairer, in Scalable TCC, we modify the protocol so that the address set is sent as a signature (like the other protocols).

Processor & Interconnect	Cache Subsystem
Cores: 64 in a multicore	Private write-back D-L1:
Signature:	Size/assoc/line:
Size: 2 Kbits	32KB/4-way/32B
Organization: Like in [7]	Round trip: 2 cycles
Max active chunks/core: 1 (or 2)	MSHR: 8 entries
Chunk size: 2000 instructions	Shared write-back L2:
Interconnect: 2D mesh	Size/assoc/line of local bank:
Interconnect hop latency: 7 cycles	256KB/8-way/32B
Coherence protocol: BulkCommit	Round trip local: 8 cycles
Memory roundtrip: 300 cycles	MSHR: 64 entries

Table 3: Simulated system configuration.

We execute 11 SPLASH-2 and 7 PARSEC applications running with 64 threads. Their input sets are shown in Table 5. The applications are run to completion.

7.2 Performance Comparison

Figures 6 and 7 show the execution time of the SPLASH-2 and PARSEC applications, respectively, on BC, BC-SQ, SB and ST for 64 processors. For uniformity, we only allow one active chunk per processor in all the schemes; we consider two active chunks per processor in the next section. For each application, the execution time is normalized to that of BC. Each bar is labeled with the name of the application and the protocol. The last four bars show the average. The bars are broken down into the following categories: cycles stalling waiting for a chunk to commit (*Commit*), cycles wasted due to chunk squashes (*Squash*), cycles stalling for cache misses (*CacheMiss*), and rest of the cycles (*Useful*).

For SPLASH-2 applications (Figure 6), we observe that BulkCommit is faster than ScalableBulk, Scalable TCC, and BulkCommit minus IntelliSquash. On average, BulkCommit reduces the execution time by 27% and 19% compared to Scalable TCC and ScalableBulk, respectively. For almost all the applications, BulkCommit reduces the commit and squash times. We observe only a very small commit overhead remaining in BulkCommit. One exception is Radiosity where, as we will see, a chunk commit often accesses many directories. Such a large directory group size takes longer to coordinate and can increase the chance of being preempted, which can result in slower commit. For a large directory group size, ScalableBulk shows substantial commit overhead, greater than Scalable TCC. It is because the Grouping stage of ScalableBulk is sequential.

Name	Protocol
BC	BulkCommit
BC-SQ	BulkCommit minus IntelliSquash
SB	ScalableBulk [17]
ST	Scalable TCC [8]

Table 4: Simulated cache coherence protocols.

App.	Input Set
Barnes	32768 123 0.025 0.05 1.0 2.0 5.0 0.075 0.25
Cholesky	-s tk23.O
FFT	-s -m16
FMM	two cluster plummer 16384 1e-6 5 .025 0.0 cost zones
LU	-n512 -b16
Ocean	-n514 -e1e-07 -r20000 -t28800
Radiosity	-batch -room
Radix	-n262144 -r1024 -m524288
Raytrace	car
Water-nsqr.	1.5e-16 1024 3 6 -1 3000 3 0 64 6.212752
Water-spatial	1.5e-16 1024 3 6 -1 3000 3 0 64 6.212752
PARSEC	simmedium

Table 5: Applications and input sets.

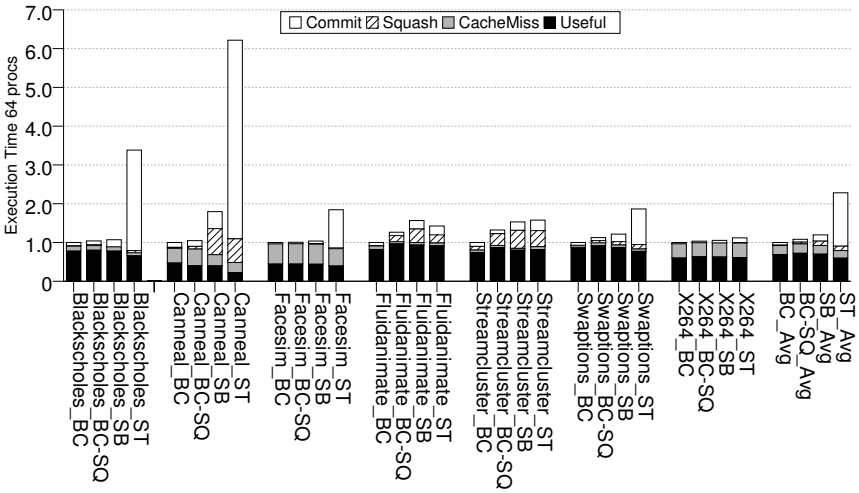

Figure 7: PARSEC execution time with 64 processors.

For PARSEC applications (Figure 7), BulkCommit reduces the execution time by 56% and 16% compared to Scalable TCC and ScalableBulk, respectively. With BulkCommit, the commit overhead practically disappears. Commit overhead is especially high for Scalable TCC. It is mainly due to contention between multiple chunks with common directories that are trying to commit at the same time. In these cases, the commits need to be completely serialized, even if the addresses accessed by the chunks do not overlap. Such effect is observable in Blackscholes and Canneal. ScalableBulk and BulkCommit allow the chunks to commit concurrently if they have overlapping directories but no overlapping line addresses.

Finally, we consider BulkCommit minus IntelliSquash (BC-SQ). This architecture substantially reduces the execution time of the applications relative to Scalable TCC and ScalableBulk. However, it still suffers noticeably from squashes in a few applications. These applications are Barnes, Ocean, Radix and Streamcluster. The squash time present in these applications with BC-SQ is mainly due to false sharing effects. For example, Radix implements a parallel radix sort algorithm that ranks integers and writes them into separate buckets for each digit. The writes to these buckets create false sharing.

Overall, on average for all the SPLASH-2 and PARSEC applications, BulkCommit reduces the execution times by 40% and 18% compared to Scalable TCC and ScalableBulk, respectively. In addition, both parts of BulkCommit are needed: while IntelliCommit is the core of BulkCommit, IntelliSquash is effective at reducing the squashes for a number of applications.

7.3 Performance with Two Active Chunks

This section evaluates the performance when each of the protocols allows a processor to have up to two active chunks at a time. Having two chunks hides commit latency, since while the first chunk is committing, the second one continues executing. However, as indicated in Section 6, after the first chunk has written to a line, the second chunk will stall if it attempts to write to any line in the same cache set (due to the Set Restriction [6]).

In the one-chunk scenario, the commit latency is fully exposed to the critical path of the execution. With two active chunks, the second chunk will hide the latency unless it is stalled by one of two reasons. First, it may stall because, when it finishes, the first chunk is not committed yet (*commitStall*). Second, it may stall due to the Set Restriction when it is about to write a line (*writeStall*).

Figure 8 characterizes the two reasons for stalling a second chunk. There is a chart for the BC, SB, and ST protocols, which correspond to the average of all the applications running with 64 processors. Each chart shows the distribution of the duration of the stall in the second chunk. Each chart has two curves: one for *writeStall* and another for *commitStall*. The second curve is not visible because there is nearly no *commitStall*. There is, however, *writeStall* in chunks.

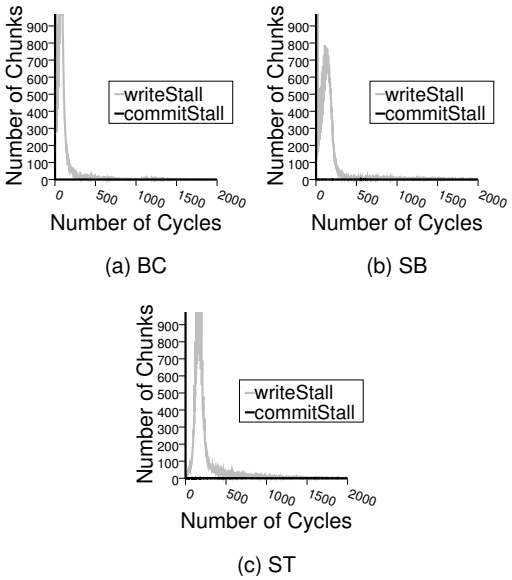

Figure 8: Distribution of the stall time for the second chunk.

From the figure, we see that the average *writeStall* for BC, SB, and ST is 97, 176, and 245 cycles, respectively. It is interesting to compare this number to the *commitStall* for the same protocols with only one active chunk per processor. It can be shown that such number is 130, 406, and 2225 cycles for BC, SB, and ST, respectively. In this case, there is no *writeStall* time. Overall, we conclude that, while supporting two active chunks per processor can hide some commit latency, it is common for the second chunk to be stalled due to *writeStall* anyway, hence negating some of the advantages of the two-chunk support.

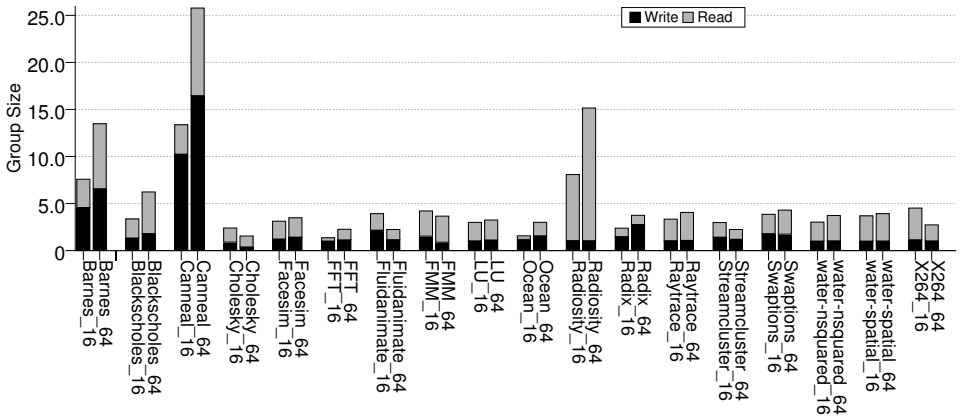

Figure 9: Directory group size.

Figure 10 compares the average execution time of the applications in all the protocols for a single active chunk and for two active chunks. The bars are labeled *1Chunk* and *2Chunks*, respectively, and correspond to the average of the applications. The figure has a chart for SPLASH-2 (a) and one for PARSEC (b). In each chart, the bars are normalized to BC with one chunk, and broken down into the usual categories.

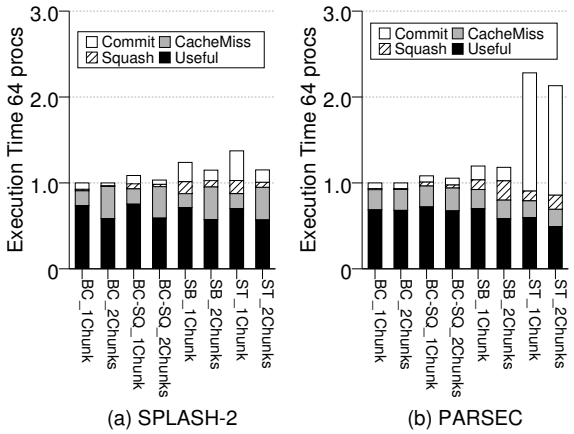

Figure 10: Performance with one and two active chunks.

We see that the use of two active chunks manages to reduce some of the commit overhead, and speeds-up the execution, relative to the single-chunk case. However, there is still some commit latency exposed to the critical path with two chunks. As we move to BC, there is practically no difference between the execution time with one and two active chunks. Given that having one active chunk is cheaper, we recommend having only one.

The bars corresponding to two active chunks per processor for SB and ST are slighty different from those for the same protocols in [17]. In particular, SB still has some commit time overhead, unlike what was shown in [17]. The reason is that, in this paper, we have used larger, more standard, input sets for the applications (Table 5) and have run the applications to completion.

7.4 Network Messages at Commit

Figure 11 shows the number of network messages at commit time in the different protocols. The figure shows bars for the average SPLASH-2 (*SpAvg*) and PARSEC (*ParAvg*) application running with 64 processors. Each bar is divided into messages containing address sets in signatures (*CommitLarge*), and other, small messages (*CommitSmall*). The SPLASH-2 bars are normalized to *SpAvg_BC*, while the PARSEC ones to *ParAvg_BC*. As a reference, commit messages account for only 6% and 4% of all the network messages in *SpAvg_BC* and *ParAvg_BC*, respectively. The other network messages are the regular execution traffic, which is roughly the same in all of the protocols.

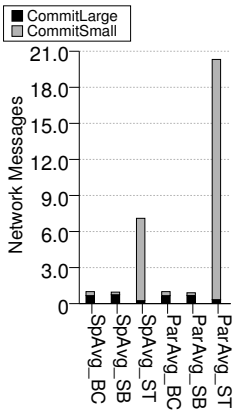

Figure 11: Number of network messages at commit.

From the figure, we see that BulkCommit and ScalableBulk have a similar number of commit messages. Given that they account for 4–6% of all the messages, they are not likely to contribute much to the execution overhead. However, Scalable TCC creates a significant number of small messages at commit. They are small broadcast messages such as Skip. It can be shown that, compared to the total number of messages in Scalable TCC, commit messages are now 31% and 46% of all messages in *SpAvg_ST* and *ParAvg_ST*, respectively. Consequently, message contention is likely to contribute to the *Commit* time in Figures 6 and 7.

7.5 Directory Group Size

Figure 9 shows the average number of directories accessed on a chunk commit for each of the applications. These are the directories that map data accessed by the chunk. Each bar is divided into Read and Write directories. A Write directory is one that maps at least one of the data written in the chunk; a Read directory only maps data read. The figure shows data for 64-processor runs and, for reference, also 16-processor runs.

From the figure, we see that most applications have a group size of around 4 directories. A few applications, such as Barnes, Can-

neal and Radiosity have large groups. However, we have shown that BulkCommit performs well even for them. For 16-processor runs, the average directory group size is typically smaller, as expected. There are a few applications where the average group size goes slightly up due to unusual data mapping effects.

8. RELATED WORK

Beyond the distributed commit protocols in directory-based lazy environments listed in Section 2, the most relevant work to IntelliSquash includes techniques that try to increase the concurrency of conflicting transactions in hardware transactional memory. There are three proposals, namely DATM [19], SONTM [2], and BulkSMT [18]. These systems use conflict serialization to avoid squashing dependent transactions (chunks). However, these systems can add considerable complexity. For example, DATM [19] needs 11 more stable states beyond the basic MSI protocol, and additional transient states may be needed in the implementation. In BulkCommit, we have three states (H, R, G) for each chunk commit transaction, rather than for each cache line. In addition, the ordering provided by the previous three proposals may eventually cause more squashes. For example, if two chunks have cyclic WAW dependences, both chunks will be squashed. IntelliSquash avoids these squashes.

Several hybrid TM systems [20, 23] move some operations (e.g., conflict detection) out of the Grouping stage in our model; hence, they do not perfectly fit our model.

9. CONCLUSION

Architectures that continuously execute chunks can improve performance and programmability. However, in a large manycore with directory-based coherence and lazy conflict detection, chunk commit can be a major bottleneck.

This paper has introduced BulkCommit, a scheme that provides scalable and fast chunk commit for these systems. The paper made three contributions. First, it presented a novel model of chunk commit in these machines. Second, it introduced the two new, general techniques that, together, constitute BulkCommit. One is the serialization of the write sets of output-dependent chunks, which eliminates chunk squashes due to WAW conflicts. The other is the parallelization of how the committing chunk grabs directory ownership, which speeds-up the commit's critical path. Our results with PARSEC and SPLASH-2 codes for 64 processors show that BulkCommit eliminates most of the squash and commit stall times. Codes run an average of 40% and 18% faster than on previously-proposed schemes.

10. REFERENCES

[1] W. Ahn, S. Qi, J.-W. Lee, M. Nicolaides, X. Fang, J. Torrellas, D. Wong, and S. Midkiff. BulkCompiler: High-performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *International Symposium on Microarchitecture*, Dec. 2009.

[2] U. Aydonat and T. Abdelrahman. Hardware Support for Relaxed Concurrency Control in Transactional Memory. In *International Symposium on Microarchitecture*, 2010.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *Int. Symposium on Computer Architecture*, 2009.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *International Symposium on Computer architecture*, 2007.

[5] E. Borin, Y. Wu, C. Wang, and M. Breternitz. LAR-CC: Large Atomic Regions with Conditional Commits. In *International Symposium on Code Generation and Optimization*, April 2011.

[6] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *International Symposium on Computer Architecture*, June 2006.

[7] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *International Symposium on Computer Architecture*, June 2007.

[8] 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 *International Symposium on High Performance Computer Architecture*, February 2007.

[9] R. Das, S. Eachempati, A. Mishra, V. Narayanan, and C. Das. Design and Evaluation of a Hierarchical On-chip Interconnect for Next-generation CMPs. In *International Symposium on High Performance Computer Architecture*, February 2009.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, March 2009.

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *International Symposium on Computer Architecture*, June 2004.

[13] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Int. Conference on Supercomputing*, July 1998.

[14] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*, June 2008.

[15] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *International Symposium on Computer Architecture*, June 2007.

[16] S. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2008.

[17] X. Qian, W. Ahn, and J. Torrellas. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *International Symposium on Microarchitecture*, December 2010.

[18] X. Qian, B. Sahelices, and J. Torrellas. BulkSMT: Designing SMT Processors for Atomic-Block Execution. In *International Symposium on High-Performance Computer Architecture*, February 2012.

[19] H. Ramadan, C. Rossbach, and E. Witchel. Dependence-Aware Transactional Memory for Increased Concurrency. In *International Symposium on Microarchitecture*, 2008.

[20] A. Shairaman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *International Symposium on Computer Architecture*, June 2008.

[21] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, June 1995.

[22] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *International Symposium on High-Performance Computer Architecture*, February 1998.

[23] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-Lazy Hardware Transactional Memory. In *Int. Symposium on Microarchitecture*, 2009.

[24] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *Communications of the ACM*, 52(12), 2009.

[25] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *Int. Conf. on Pervasive Systems*, July 2005.

[26] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *International Symposium on Computer Architecture*, June 2007.