ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment*

Xuehai Qian, Wonsun Ahn, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Recently-proposed architectures that continuously operate on atomic blocks of instructions (also called *chunks*) can boost the programmability and performance of shared-memory multiprocessing. However, they must support chunk operations very efficiently. In particular, in lazy conflict-detection environments, it is key that they provide scalable chunk commits. Unfortunately, current proposals typically fail to enable maximum overlap of conflict-free chunk commits.

This paper presents a novel directory-based protocol that enables highly-overlapped, scalable chunk commits. The protocol, called *ScalableBulk*, builds on the previously-proposed BulkSC protocol. It introduces three general hardware primitives for scalable commit: preventing access to a set of directory entries, grouping directory modules, and initiating the commit optimistically. Our results with SPLASH-2 and PARSEC codes with up to 64 processors show that ScalableBulk enables highly-overlapped chunk commits and delivers scalable performance. Unlike previously-proposed schemes, it removes practically all commit stalls.

1. Introduction

There are several recent proposals for shared-memory architectures that efficiently support continuous atomic-block operation [2, 5, 6, 8, 9, 14, 18, 19]. In these architectures, a processor repeatedly executes blocks of consecutive instructions from a thread (also called *chunks*) in an atomic manner. These systems include TCC [6, 9], BulkSC [5], Implicit Transactions (IT) [18], ASO [19], InvisiFence [2], DMP [8], and SRC [14] among others. This mode of execution has performance and programmability advantages. For example, it can support transactional memory [6, 9, 14]; high-performance execution, even for strict memory consistency models [2, 5, 19]; a variety of techniques for parallel program development and debugging such as determinism [8], program replay [12], and atomicity violation debugging [10]; and even provide a substrate for new high-performance compiler transformations [1, 13].

For these machines to deliver scalable high performance, the cache coherence protocol must support chunk operations very efficiently. It must understand and operate with chunk transactions like conventional machines operate with individual memory accesses.

There are several ways to design a chunk cache coherence protocol. In particular, an important decision is whether to use lazy or eager detection of chunk conflicts. In the former, chunks execute obliviously of each other and only check for conflicts at the time of chunk 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 [3, 17] — although there is no consensus on which approach is the most promising one. In this paper, we focus on an environment with lazy conflict detection [5, 6, 8, 9, 14, 18].

In such an environment, a major bottleneck is the chunk commit operation, where the system checks for collisions with all the other executing chunks. Early proposals used non-scalable designs. For example, TCC [9] relies on a bus, while BulkSC [5] uses a centralized arbiter. Later designs such as Scalable TCC [6] and SRC [14] work with a directory protocol and, therefore, are more scalable. However, Scalable TCC requires broadcasting and centralized operations, and SRC has message serialization. More importantly, these schemes add *unnecessary commit serialization* by disallowing the concurrent commit of chunks that, while collision-free, happen to use the same directory module(s). This problem is worse for applications with *lower locality* and for *higher processor counts*.

To address this problem, this paper presents a novel directory-based protocol that enables highly-overlapped, scalable commit operations for chunks. In our design, the commit operation uses no centralized structure and communicates only with the relevant directory modules. Importantly, it enables the concurrent commit of any number of chunks that use the same directory module — as long as their addresses do not overlap. Our goal is to emulate for chunks what conventional directories do for individual write transactions.

The protocol, called *ScalableBulk*, builds on the previously-proposed BulkSC protocol — effectively extending it to work with directories. ScalableBulk introduces three new generic hardware primitives for scalable chunk commit: (1) preventing access to a set of directory entries, (2) grouping directory modules, and (3) initiating the commit optimistically.

We evaluate ScalableBulk with simulations running SPLASH-2 and PARSEC codes with up to 64 processors. Our results show that ScalableBulk enables highly-overlapped chunk commit operations and delivers scalable performance. For 64-processor executions, ScalableBulk does not suffer from practically any commit stall overhead. We show that this is unlike the previously-proposed schemes.

This paper is organized as follows: Section 2 describes our goal and related work; Section 3 presents the ScalableBulk protocol; Section 4 describes implementation issues; and Sections 5 and 6 evaluate ScalableBulk.

2. Building a Lazy Scalable Chunk Protocol

In a chunk cache coherence protocol that performs lazy conflict detection, the chunk commit operation is critical. Indeed, during the execution of a chunk, cache misses bring individual lines into the cache, but no write is made visible outside the cache. At commit, the changes in coherence states induced by all the writes must

* This work was supported in part by the National Science Foundation under grant CNS 07-20593; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); and Sun Microsystems under the UIUC OpenSPARC Center of Excellence.

be made visible to the rest of coherent caches atomically — squashing any other chunk that has a data collision. Note that a commit does not involve writing data back to memory. However, it requires updating the states of the distributed caches in a way that appears that chunks executed in a total order.

Architectures that support continuous chunk operation in this environment [5, 6, 8, 9, 14, 18] must critically provide efficient chunk commit. In general, while they must commit dependent chunks serially, they attempt to overlap the commit of independent chunks. In the following, we describe the main existing proposals for concurrent commits, reconsider whether commit is critical, and describe our approach.

2.1. Main Proposals for Concurrent Commits

BulkSC [5] uses a centralized arbiter that receives every commit request. The arbiter allows the concurrent commit of multiple chunks, as long as the addresses that an individual chunk wrote do not overlap with the addresses accessed by any other chunk. To detect overlap at a fine grain, BulkSC uses hardware address signatures [4].

Scalable TCC [6] supports chunk commits in a directory-based coherence protocol. The protocol overlaps the commit of independent chunks but has several scalability bottlenecks. First, the committing processor contacts a centralized agent to obtain a transaction ID (TID), which will enforce the order of chunk commit. Second, the processor contacts *all* the directory modules in the machine — each one possibly multiple times. Specifically, it sends a *probe* message to the directories in the chunk's write- and read-set, and a *skip* message to the rest. Third, for every cache line in the chunk's write-set, the processor sends a *mark* message to the corresponding directory.

More importantly, however, is that this protocol limits the concurrent commit of independent chunks. Indeed, two chunks can only overlap their commits if they use *different* directories. In other words, if two chunks access *different* addresses but those addresses are in the *same* directory module, their commit gets serialized.

SRC [14] focuses on removing the TID centralization and the message multicasts from Scalable TCC. The idea is for each committing processor to send messages to the directories in the chunk's read- and write-sets to sequentially occupy them. For example, if the chunk's sets include directories 1, 4 and 6, the processor starts by occupying 1, then 4, then 6. The transaction gets blocked if one directory is taken. This protocol, called SEQ-PRO, reduces centralization. However, it introduces sequentiality. Importantly, it has the same shortcoming as Scalable TCC: two chunks that accessed different addresses from the same directory are serialized.

The authors present an optimization called SEQ-TS where the committing processor sends a request in parallel to all the directories in its read- and write-sets, and can steal a directory from the chunk that currently occupies it. However, this approach seems prone to protocol races, and there are little details on how it works.

2.2. Is Commit Really Critical?

There are two papers that show experimental data on the scalability of these protocols. One is the Scalable TCC paper [6], which shows simulations of SPEC, SPLASH-2, and other codes for 64 processors. The other is the SRC paper [14], which shows simulations of synthetic traces for 64-256 processors. The data in both papers appears to suggest that chunk commit is overlapped with computation and does not affect the execution time of applications.

However, the environments described in these papers are different from the one we are interested in. There are two key differences: the size of the chunks and the number of directories accessed per chunk commit.

Consider the chunk sizes first. The chunks (i.e., transactions) in Scalable TCC are large, often in the range of 10K-40K instructions. Such large chunks are attained by manually (or automatically) instrumenting the application, positioning the transactions in the best places in the code. In SRC, the authors consider synthetic models with chunks larger than 4K instructions.

We are interested in an environment like BulkSC, where the code is *executed as is*, without software instrumentation or analysis. Following BulkSC, we use chunk sizes of 2K instructions. Additionally, cache overflows and system calls can further reduce the average size. With chunk sizes that are one order of magnitude smaller than Scalable TCC, chunk commit is more frequent, and its overhead is harder to hide.

Consider now the number of directories accessed per chunk commit operation. In Scalable TCC, for all but two codes, the 90th percentile is 1–2 directories. This means that, in 90% of the commits, only one (or at most two) directories are accessed. In SRC, the synthetic model is based on the Scalable TCC data.

This is in contrast to the larger numbers that we observe in our evaluation. In this paper, the average number of directories accessed per chunk commit is typically 2–6. We believe the difference is because, in our environment, we cannot tune what code goes into a chunk. With these many directories, we often concurrently commit chunks that accessed disjoint addresses but use the same directory. Scalable TCC and SRC would serialize them.

Overall, from this discussion, we conclude that the commit operation is indeed time-critical.

2.3. Our Approach

We devise a directory-based coherence protocol that works with chunks efficiently. A truly scalable chunk commit operation should (i) need no centralized structure, (ii) communicate only with the relevant directories, and (iii) allow the concurrent commit of chunks that use the same directory, as long as their addresses do not overlap. Moreover, we believe that hardware address signatures [4, 11, 20] provide a good means to implement a chunk protocol efficiently. They perform operations on footprints inexpensively.

3. The ScalableBulk Protocol

We describe ScalableBulk by focusing on its three new generic protocol primitives: (1) preventing access to a set of directory entries, (2) grouping directory modules, and (3) initiating the commit optimistically. In our discussion, we assume a multicore architecture with distributed directory modules as in Figure 1.

3.1. Preventing Access to a Set of Directory Entries

In a chunk protocol like the one considered, there are no individual write transactions. Instead, all the writes in a chunk are processed with a single commit transaction at the directory. To understand such a transaction, consider the operation of a write in a conventional directory protocol. When a write arrives at a directory, the controller starts a transaction that involves (in a four-hop protocol): setting the directory state for the line to transient, identifying the sharers, sending invalidations to the sharers, receiving acknowledgments (acks), updating the directory state for the line to dirty,

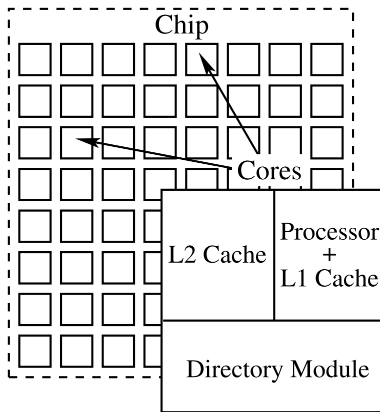


Figure 1. Generic multicore architecture considered.

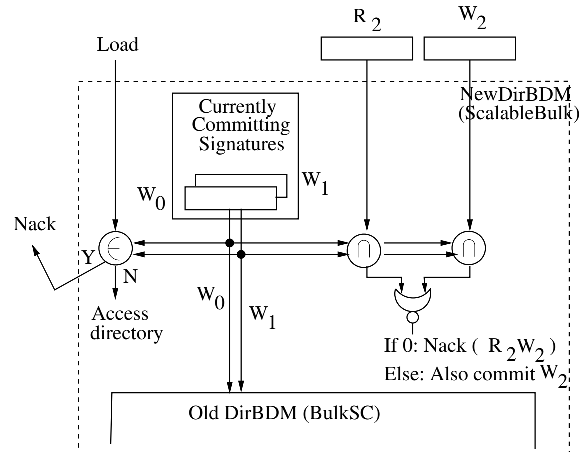


Figure 2. A ScalableBulk directory module allows the commit of multiple, non-overlapping chunks, and nacks overlapping accesses and overlapping chunk commits.

and notifying the writer. While the state is transient, the directory controller blocks all requests to the same line — either by buffering the requests or by bouncing them (i.e., nacking). However, at all times, the controller can process transactions for other lines.

In a chunk protocol, there is a single transaction for each chunk commit. Let us assume for now that the machine has a single directory module. When the directory receives the commit request, the controller must identify the addresses of the lines written by the chunk. In ScalableBulk, this is done by expanding the write (W) signature [5] (while in other schemes, the controller may receive the list of written addresses). Then, the controller compiles the list of sharers of such lines, sends W to them for cached line invalidation and chunk disambiguation, and finally collects all acks. In the meantime, the directory controller updates the state of the directory entries for these lines.

During this process, and until all acks are received and the directory state for all of these lines is updated, the directory controller must disable access to these lines. It does so by nacking both (i) reads to the lines and (2) commits of chunks that have read or written these lines. However, commits (and reads) that do not have any address overlap with these lines should proceed in parallel. Moreover, the decision of whether or not to nack should be quick.

Figure 2 shows how these operations are performed in ScalableBulk. In the figure, two chunks given by W signatures W_0 and W_1 are committing concurrently. Signature expansion is performed in a module like the DirBDM in BulkSC [5]. Any incoming load to the directory module is checked for membership in W_0 and W_1. If there is no match, the access is allowed to proceed. Any incoming read/write signature pair (R_2, W_2) for a chunk is intersected with W_0 and W_1. If all the intersections are null, W_2 is allowed to join W_0 and W_1 in committing. Note that all these operations are fast. Moreover, false positives due to signature aliasing cannot affect correctness. At worst, they nack an operation unnecessarily.

3.2. Grouping Directory Modules

In a conventional directory protocol, a write access engages a single directory module; in a chunk protocol, a chunk commit can require the participation of multiple directories — the home directories of the lines read or written in the chunk. Coordinating the multiple directories in a commit is the biggest challenge in any chunk protocol.

For commit scalability, ScalableBulk only communicates with the home directories of the lines read or written in the chunk — rather than with all the directories as in Scalable TCC. Coordinating multiple directories is done in two steps, namely identifying the directories and then synchronizing their operation in what we call *Directory Grouping*. To identify the relevant directories, the hardware could encode the signatures in a way that made it easy to extract the home directory numbers of the constituting addresses. However, this approach is likely to require a non-optimal signature encoding. Consequently, ScalableBulk works differently: as a chunk executes, the hardware automatically collects in a list the home directory numbers of the reads and writes issued. At chunk commit time, the compressed R and W signatures and this list are sent to the directory modules in the list.

For any group of directories that receive a (R, W) signature pair, ScalableBulk designates a *Leader*. The leader is set by a simple, default hardware policy. The baseline policy is for the leader to be the lowest-numbered module in the group. The leader initiates a synchronization step using the *Group Formation* protocol (*Group* for short). The protocol, which is described next, attempts to group all participating directories. If it succeeds, the leader sends a *commit success* message to the committing processor, which then clears its signatures; if it fails, the leader sends a *commit failure* message to the committing processor, which prompts it to wait for a while and then retry the commit request (unless the committing processor is asked to squash the chunk before).

As the leader sends the *commit success* message, it sends the W signature to all the sharer processors, to trigger cached line invalidation and chunk disambiguation. Later, when the leader receives all acks, it multicasts a *commit done* message to the directory group. The directories in the group silently break down the group and discard W.

From the time a directory receives the (R, W) signature pair and tries to form a group until it receives the *commit done* message (or the group formation fails), it nacks incoming overlapping requests and overlapping commits.

3.2.1. Group Formation Protocol

At any time, there may be multiple sets of directory modules trying to form groups. Some of these groups may be incompatible with each other. Two groups that are trying to use a given directory module are *Incompatible* if their W signatures overlap or if the R

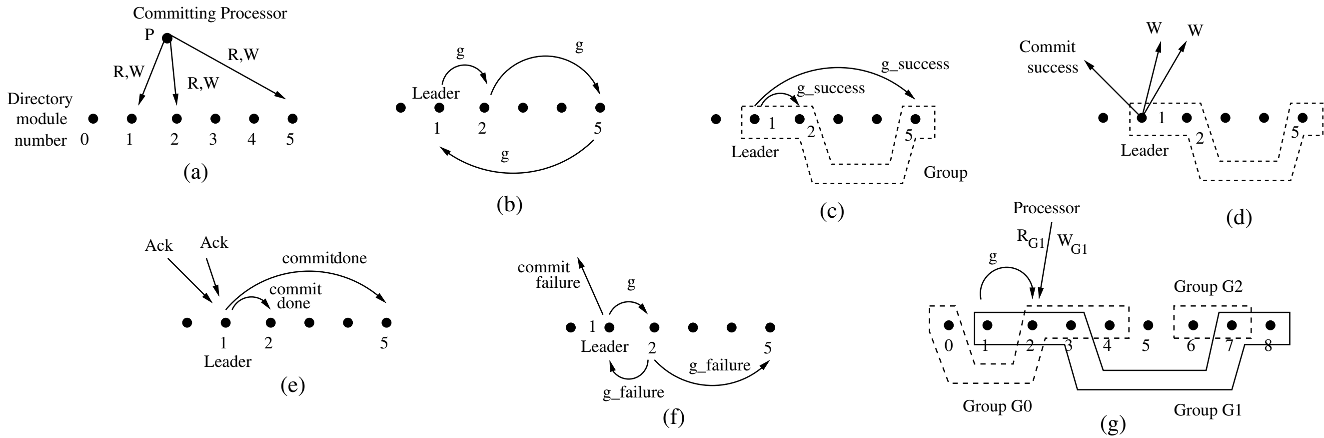


Figure 3. Group formation protocol.

signature of one and the W signature of the other overlap. Otherwise, they are compatible and can commit concurrently.

The Group protocol ensures that: (i) all the compatible groups form successfully concurrently, and (ii) given a set of incompatible groups, at least one of them forms. To guarantee deadlock- and livelock-freedom, the Group protocol follows well-known deadlock-free resource-allocation algorithms by requiring a fixed directory-module traversal order. Specifically, the algorithm forms a group by always starting from the leader and traversing directory modules in hardware from lower to higher numbers.

The algorithm is shown in Figure 3, which depicts six directory modules. All arrows are hardware messages. In Chart (a), a committing processor sends the (R, W) signature pair to participating directories 1, 2, and 5. The message also contains the list of participating directories. Each of these directory modules expands the W signature and, after checking its local directory state, determines the list of processors that need to be invalidated to commit the chunk (namely, the sharer processors). In addition, the leader starts by putting its list in a *g* (or *grab*) message and sending it to the next directory module in the sequence (Chart (b)). Each directory module, when it receives *g*, if it has already received the signatures and expanded W to find the sharer processors, augments the processor list in *g*, and passes the updated *g* to the next directory in the sequence. Note that computing the sharer processors is done by all directory controllers in parallel, typically before they receive the *g* message. Therefore, it is not in the critical path.

Eventually, the *g* message returns to the leader. The leader then multicasts a short *g_success* message to all participating directory modules (Chart (c)). The group is now formed, and the directories start updating their state based on the W signature. At the same time, the leader sends a *commit success* message to the committing processor and W to the sharer processors (Chart (d)). On reception of all the acks from the sharers, the leader multicasts a *commit done* message to the directory group (Chart (e)). All the directories in the group then break down the group and deallocate the W signature.

It is possible that the *g* message does not return to the leader. This occurs when the group being formed collides with a second group and the latter wins. The collision occurs in the *lowest-numbered* directory module that is common to both groups. We call it the *Collision* module. It declares, as the winner group, the first group for which it sees both (i) the (R, W) signature pair coming from the committing processor and (ii) the *g* message coming from the previous directory module in the group.

As soon as the Collision node sees both messages from one group, it pushes the *g* message to the next node in that group, irrevocably choosing that group as the winner. Later, when it receives both messages from the losing group, it simply multicasts a *g_failure* message to all the directories in the losing group. The directories then deallocate the losing W signature and the leader of the losing group sends a *commit failure* message to the committing processor. Chart (f) shows this case assuming that module 2 detected the collision.

3.2.2. Forward Progress, Starvation, and Fairness

The Group algorithm guarantees forward progress because, when several groups collide, at least one is able to commit successfully. This is guaranteed because *g* messages are strictly passed from lower- to higher-numbered modules, and they are only sent when the sender has received both (R, W) and *g*.

As an advanced example, Figure 3(g) shows a system with nine directory modules and three colliding groups. The latter are G0 (which tries to combine directories 0, 2, 3, and 4), G1 (trying directories 1, 2, 3, 7, and 8), and G2 (trying directories 6 and 7). The Collision module for Groups G0 and G1 is Module 2 — the lowest-numbered, common module. Suppose that Module 2 receives the combination of (R, W) and *g* for Group G1 first. At that point, Module 2 passes *g* for G1 to Module 3, effectively choosing G1 over any future colliding group. Later, when Module 2 receives the combination of (R, W) and *g* for G0, it multicasts *g_failure* for G0 to Modules 0, 3, and 4. The next decision occurs in Module 7. The module chooses between G1 and G2 based on which of the groups first provides (R, W) and *g*. Overall, *at least* one group will form successfully.

The algorithm described tends to favor small groups over large ones. This is because large groups are likely to encounter more Collision modules as they form and, therefore, have higher chances of failing to form. To prevent the commit starvation of such chunks, the Group algorithm works as follows. After a given directory module has seen the squash of a given chunk commit for a total of *MAX* times, then, it reserves itself for the commit of that chunk. It responds to all other requests for commit as if there was a collision and the requester lost. It does this until it receives the request from the starving chunk and commits it. Since all the directories in the group see every single squash of the group, they all reserve themselves for the starving chunk at the same time.

The algorithm also tends to favor the commit of chunks from processors close to low-numbered directories. This is because these processors can push signatures to low-numbered directories faster, hence pre-empting other processors. To solve this problem and ensure long-term fairness, the Group algorithm can change the relative priority of the directory-module IDs at regular intervals. Specifically, it can use a modulo rotation scheme where, during one interval, the highest-to-lowest priority is given by IDs 0, 1, ... n; during the next interval, it is given by 1, 2, ... n, 0; and so on. At any time, the Group algorithm chooses the leader of a group to be the one with the highest-priority ID in the group, and the *g* messages are sent from higher to lower priority modules.

3.3. Optimistic Commit Initiation

In existing chunk protocols such as BulkSC, the commit operation proceeds conservatively. Specifically, the processor sends a *permission to commit* request to an arbiter and, while the processor is waiting for an *OK to commit* or *Not OK to commit* message from the arbiter, it nacks all incoming messages — such as signatures for cache line invalidation and chunk disambiguation. This action limits concurrent commit.

In a machine with many cores, communication latencies may be high, and determining whether a chunk can commit takes some time. In ScalableBulk, it takes the time to form (or fail to form) a group.

To address this issue, ScalableBulk introduces *Optimistic Commit Initiation (OCI)*, where a committing processor assumes that its commit transaction will succeed. After the processor sends its commit request with signatures to the target directories, it continues to consume incoming messages — including signatures from concurrently committing transactions that attempt to perform bulk invalidation (i.e., invalidate cached lines and disambiguate against the local chunk). Note that the local chunk's R and W signature registers are not deallocated until the processor receives a *commit success* message and, therefore, are available for disambiguation.

OCI increases performance by increasing the overlap of multiple commits. Moreover, by doing so, it also reduces the time that signatures are buffered in directory modules (Section 3.1). This in turn reduces the time during which requests and signatures are nacked from directories, and decreases the possibility of collisions.

However, OCI complicates the protocol when the committing processor consumes a bulk invalidation message and finds that it needs to squash the chunk that it recently sent out for commit. In this case, as the processor squashes and restarts the chunk, it sends a *Commit Recall* message to ask for the cancellation of the commit. This recall message is piggy-backed on the ack to the bulk invalidation message that caused the squash. As we show in Section 3.4, ScalableBulk ensures that this message is propagated to the correct directories. Later, when the processor receives a *commit failure* message from the leader of its failed directory group, it discards it.

The OCI protocol is illustrated in Figure 4. Consider two processors that initiate commits with overlapping addresses. Processor P0 sends its signatures to directory modules 0, 2, and 3, while P1 sends its own to modules 1, 2, and 3 (Chart (a)). The first set of directories succeed in forming Group G0. Its leader (Module 0) sends *commit success* to P0, and W_0 for bulk invalidation to P1 (Chart (b)). At this point, a conservative protocol proceeds as in Chart (c), while one with OCI proceeds as in Chart (d).

Specifically, in Chart (c), P1 nacks the W_0 message repeatedly until it receives a *commit failure* message from the leader of the

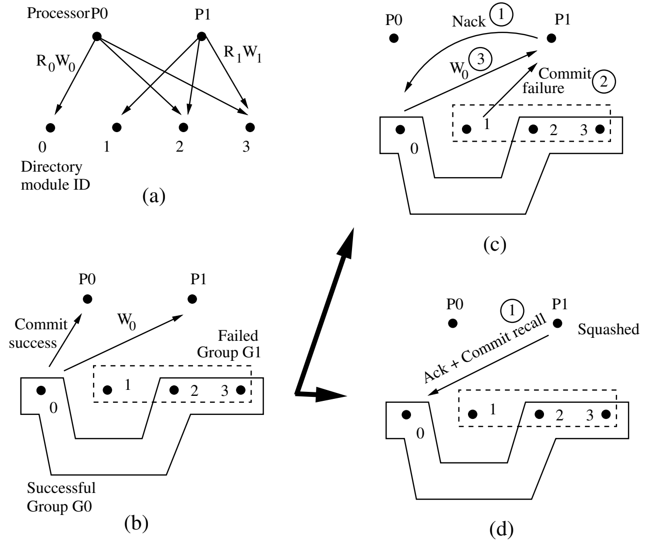


Figure 4. Operation of the Optimistic Commit Initiation (OCI).

group that failed (Group G1). At that point, it consumes the W_0 message and squashes the local chunk — therefore enabling the completion of the Group G0 chunk commit.

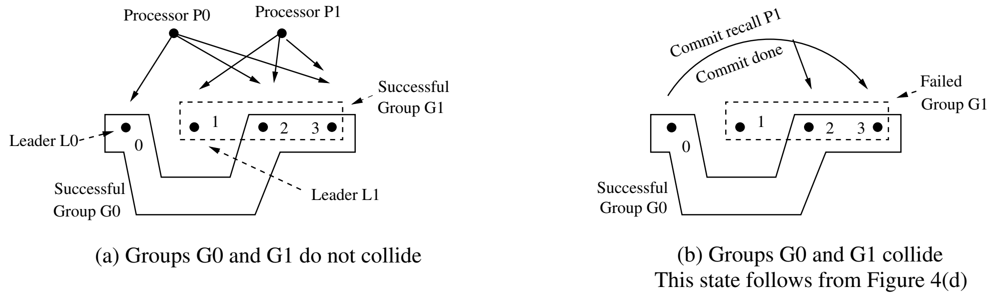
On the other hand, in Chart (d), P1 immediately consumes the W_0 message, piggy-backs a *commit recall* on the ack to the W_0 (bulk invalidation) message, and squashes and restarts its chunk. We show in Section 3.4 that this recall message is routed to the directories of Group G1, to tell them that the committing processor has squashed its chunk. Later, when P1 receives the *commit failure* message for the chunk from Module 1, it discards it.

Overall, OCI reduces the critical path to complete the successful commit of the chunk in Group G0. Specifically, it removes from the critical path the following potential latencies of failed Group G1 operation: the initial request from P1 to the directory modules participating in Group G1, the (failed) formation of Group G1, and the transfer of the *commit failure* message to P1.

3.4. Putting it All Together: Scalable Commit

The ScalableBulk features described fulfill the requirements for a truly scalable commit operation listed in Section 2.3. First, there is no centralization point. Second, a committing processor communicates only with the directory modules in its signatures, with point-to-point messages; there is no message broadcasting. Third, any number of chunks that share directory modules but have non-overlapping updated addresses ($R_i \cap W_j \vee W_i \cap W_j$ is null for every i, j pair) can commit concurrently — just as conventional protocols support any number of concurrent write transactions to different addresses. Finally, OCI maximizes the overlap of commits by removing operations from the critical path of commits.

To show how the complete chunk commit operation works, we revisit Figure 4(a), where two chunks that have accessed data from common directory modules (Modules 2 and 3) try to commit. Let us consider two cases: in one, they do not have overlapping updated addresses; in the other, they do. When they do not, they commit concurrently (Figure 5(a)). Each processor sends the signatures to the relevant directory modules. The leader module in each group forms the group. Then, each leader sends a *commit success* to its own originating processor, and bulk invalidations to all sharer processors. When the sharer processors ack, the leader multicasts a



(a) Groups G0 and G1 do not collide

(b) Groups G0 and G1 collide
This state follows from Figure 4(d)

Figure 5. Scalable chunk commit, where both commits succeed (a) and where only one does (b).

Message Type	Description	Format	Direction
commit_request	Processor requests to commit a chunk. Message is sent to all the directory modules in the read- and write-sets of the chunk	$C_Tag, W_Sig, R_Sig, g_vec$	Proc → Dir(s)
g (or *grab*)	Source directory module is part of a group, and tries to grab the destination module to put it into the same group	$C_Tag, inval_vec$	Dir → Dir
g_failure	A module detects that group formation has failed and notifies of the failure to all the modules in the group	C_Tag	Dir → Dir(s)
g_success	The leader informs all the modules in the group that the group has been successfully formed	C_Tag	Dir → Dir(s)
commit_failure	The leader notifies the commit-requesting processor that the commit failed	C_Tag	Dir → Proc
commit_success	The leader notifies the commit-requesting processor that the commit is successful	C_Tag	Dir → Proc
bulk_inv	The leader sends out a bulk invalidation to all the sharer processors	C_Tag, W_Sig	Dir → Proc(s)
bulk_inv_ack	The leader receives a bulk invalidation acknowledgment from a sharer processor	C_Tag	Proc → Dir
commit_done	The leader releases all the modules in the group and requests the deallocation of the signatures	C_Tag	Dir → Dir(s)
commit_recall	A processor with a squashed chunk notifies the leader module of the squash. The message is piggy-backed on *bulk_inv_ack* and *commit_done* messages	C_Tag, Dir_ID	Proc → Dir, Dir → Dir

Table 1. Message types in ScalableBulk.

commit done to all directory modules in the group, which deallocate the signature. These operations proceed in parallel for the two groups.

Consider now that the two chunks have overlapping updated addresses. Each processor sends the signatures to the relevant directory modules. Assume that, as shown in Figure 4(b), Group G0 succeeds and G1 fails. P1 receives the W_0 signature (i.e., the bulk invalidation message), squashes the chunk it is committing (Figure 4(d)), and piggy-backs a *commit recall* in the ack to G0's leader (Module 0).

Figure 5(b) shows the state after Figure 4(d). As Module 0 multicasts the *commit done* to the G0 members, it includes the *commit recall* from P1 in the message. All modules deallocate signature W_0 and consider the commit complete. The *commit recall* triggers no action in any module except in the lowest-numbered one of the set of modules common to both G0 and G1 (the Collision one). In our example, this is Module 2. The *recall* tells Module 2 that P1 started a commit and its chunk has already been squashed. If Module 2 has already seen both (R_1, W_1) and *g* for G1, then it has already sent *g_failure* to all the members of Group G1. Consequently, it simply discards the *commit recall*. Otherwise, the *commit recall* tells Module 2 to be on the lookout for the reception of (R_1, W_1) and *g* for G1; when it receives them both, it sends the *g_failure* message to all the G1 group members. This is why the *commit recall* is needed: Module 2 deallocates signature W_0 and, therefore, would not be able to detect the collision if it has not yet observed (R_1, W_1).

4. Implementation Issues

To get a flavor of ScalableBulk's implementation, this section describes the message types and the design of a directory module. Appendix A describes the ordering of messages in a directory module.

4.1. Message Types

Table 1 shows all of the message types needed in ScalableBulk. There are ten types. In the table, C_Tag is the unique tag assigned to a chunk. It is formed by concatenating the originating processor ID and a processor-local sequence number. W_Sig and R_Sig are the write and read signatures of a chunk. g_vec is the set of directory modules in a chunk's read- and write-sets. It is formed by the processor as it executes a chunk. $inval_vec$ is a bit vector with the set of sharer processors that need to be invalidated once a group has been formed. It is built incrementally at each participating module, and passed with the *g* message. The *commit_recall* message is piggy-backed on the *bulk_inv_ack* message and then on the *commit_done* message, so that it reaches the Collision directory module (indicated by Dir_ID). Finally, *Proc* and *Dir* mean processor and directory module, respectively.

As an example, the first row describes the request-to-commit message (*commit_request*), sent by a committing processor to all the directories in the read- and write-sets of the chunk. The message includes the chunk tag, the signatures, and the set of directories in the chunk's read- and write-sets. The other rows can be easily followed.

4.2. Directory Module Design

Figure 6 shows the design of a ScalableBulk directory module. It has three components, namely buffers for the incoming and outgoing messages, the *Chunk State Table* (CST) that contains one entry per committing or pending chunk, and the ScalableBulk protocol engine. The protocol engine implements the protocol state machine. It accepts incoming messages, triggers the state transitions for the corresponding chunks in the CST, and potentially generates new outgoing messages based on the state of the chunks. To design the state machine, we follow the methodology summarized in [16], and derive the set of states, events, messages, transitions, and actions with each transition.

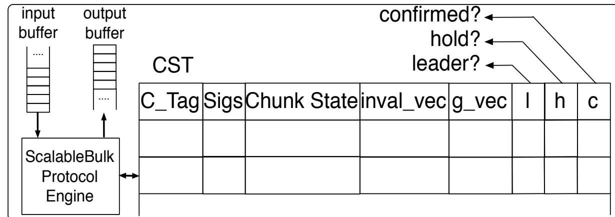


Figure 6. Directory module in ScalableBulk.

Each CST entry corresponds to a chunk being processed by this directory module. There is an analogy between a CST entry in a ScalableBulk directory and a regular entry in a conventional directory. A CST entry is allocated when the directory module receives either a signature pair (*R_Sig*, *W_Sig*) or a *g* message for a chunk. It is deallocated at one of two points: (1) the chunk commits successfully and the directory has received all *bulk_inv_ack* messages (if it is the leader) or a *commit_done* message (if it is not), or (2) the chunk commit fails and the directory receives a *commit_recall* message (if it is the Collision module) or a *g_failure* message.

A CST entry contains several fields. C_Tag is the chunk's unique tag. *Sigs* is the R and W signatures. *Chunk State* is the state of the chunk. As indicated before, g_vec is a bit vector with the set of directory modules in the chunk's read- and write-sets, while $inval_vec$ is a bit vector with the set of sharer processors that need to be invalidated (based on the state in this directory). The final $inval_vec$ is built by accumulating the $inval_vec$ fields of all participating directories through the *g* message. Finally, there are three status bits for the chunk. *l* (*leader*) indicates whether this directory is the leader of the group. *h* (*hold*) indicates that no conflicts were found in this directory and that this directory was admitted into the group. It is set right before sending a *g* message. Finally, *c* (*confirmed*) indicates that the group has been successfully formed. For the leader, it is set after a *g* message is received from the last module in the group; for a non-leader, it is set after a *g_success* message is received from the leader.

Overall, the system operates similarly to a conventional directory-based protocol, but maintains "coherence" at the granularity of chunks.

5. Evaluation Setup

We evaluate ScalableBulk using a cycle-accurate execution-driven simulator based on SESC [15]. We model a multicore system like the one in Figure 1, in which we can configure the number of cores to be 32 or 64. The cores issue and commit one instruction per cycle. Memory accesses can be overlapped with instruction execution through the use of a reorder buffer. Each core has private L1 and L2 caches that are kept coherent using a directory-based scheme that implements the ScalableBulk protocol. The cores are connected using an on-chip 2D torus modeled with the simulator of Das *et al* [7]. A simple first-touch policy is used to map virtual pages to physical pages in the directory modules. Table 2 shows more details.

Processor & Interconnect	Memory Subsystem
Cores: 32 or 64 in a multicore Signature: Size: 2 Kbits Organization: Like in [5] Max active chunks per core: 2 Chunk size: 2000 instructions Interconnect: 2D torus Interconnect link latency: 7 cycles Coherence protocol: ScalableBulk	Private write-through D-L1: size/assoc/line: 32KB/4-way/32B Round trip: 2 cycles MSHRs: 8 entries Private write-back L2: size/assoc/line: 512KB/8-way/32B Round trip: 8 cycles MSHRs: 64 entries Memory roundtrip: 300 cycles

Table 2. Simulated system configurations.

For the evaluation, we run 11 SPLASH-2 applications and 7 PARSEC applications. The applications run with 32 and 64 threads. We run the applications with reference data sets for all runs. For LU and Ocean from SPLASH-2, we use the more locality-optimized contiguous versions. For the PARSEC applications, we use the small input sets, except for Dedup and Swaptions, which run with the medium and large input sets, respectively, due to scalability reasons.

We also implement and evaluate several protocols proposed in previous work. They are shown in Table 3.

Name	Protocol
ScalableBulk	Protocol proposed
TCC	Scalable TCC [6]
SEQ	SEQ-PRO from [14]
BulkSC	Protocol from [5] with arbiter in the center

Table 3. Simulated cache coherence protocols.

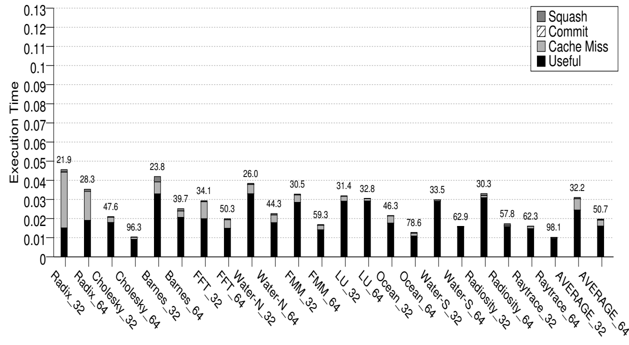
6. Evaluation

In our evaluation, we examine the performance and scalability, the number of directory modules accessed per chunk commit, the chunk commit latency, and the chunk commit serialization. We finally characterize the traffic.

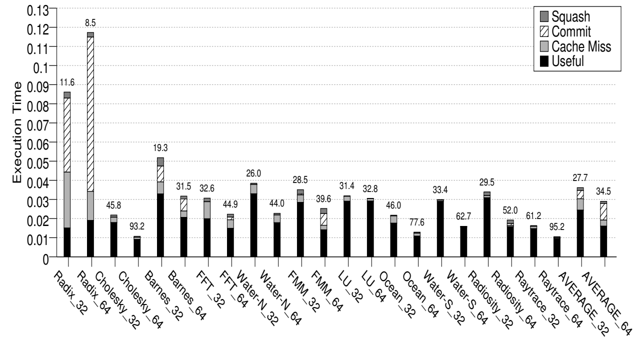
6.1. Performance and Scalability

Figures 7 and 8 show the execution time of the applications on ScalableBulk, TCC, SEQ, and BulkSC for 32 and 64 processors — normalized to the execution time of single-processor runs on the same architecture with ScalableBulk. Each bar is labeled with the name of the application and the number of processors. The last two bars show the average. The bars are broken down into the following categories, from bottom to top: cycles executing one instruction (*Useful*), cycles stalling for cache misses (*Cache Miss*), cycles stalling waiting for a chunk to commit (*Commit*) and cycles wasted due to chunk squashes (*Squash*). The number on top of each bar is the speedup.

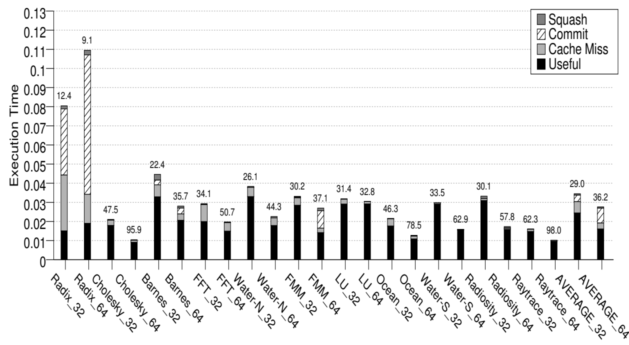
From the average values, we see that the BulkSC protocol does not scale well going from 32 to 64 processors due to its centralized nature. Distributed protocols such as ScalableBulk, TCC, and SEQ



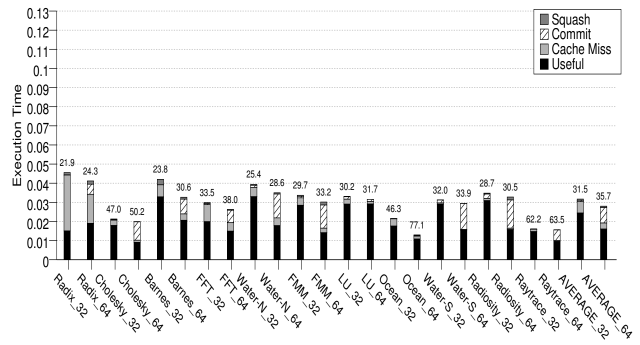
(a) ScalableBulk



(b) TCC

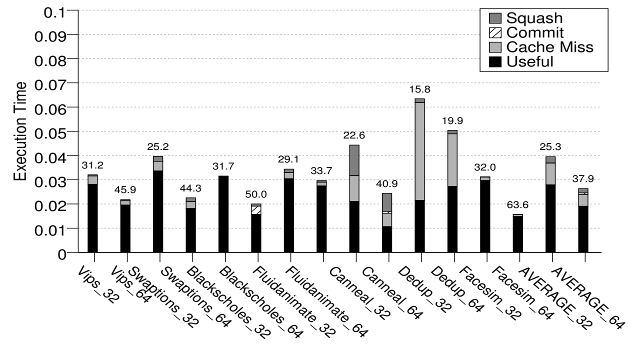


(c) SEQ

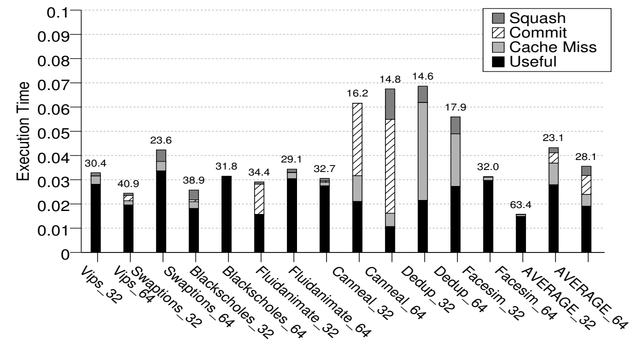


(d) BulkSC

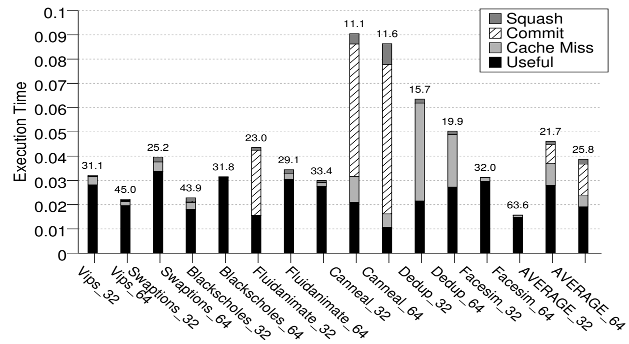
Figure 7. Execution times of the SPLASH-2 programs normalized to single-processor runs with ScalableBulk.



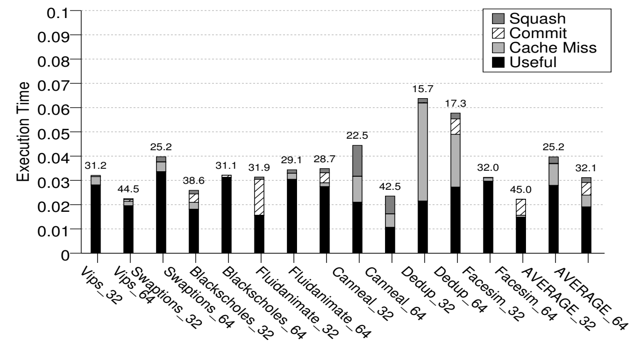
(a) ScalableBulk



(b) TCC



(c) SEQ



(d) BulkSC

Figure 8. Execution times of the PARSEC programs normalized to single-processor runs with ScalableBulk.

scale better going from 32 to 64 processors, but TCC and SEQ show significant commit overhead for applications such as Radix, Barnes, Canneal, and Blackscholes. ScalableBulk suffers almost no commit overhead even for these applications due to its overlapped nature. Squash overhead is generally minimal, since data conflicts between two chunks are relatively rare and not very costly even when they happen — given the small chunk size (2000 instructions). In ScalableBulk for 64 processors, only 1.5% of all chunks were squashed due to data conflicts and 2.3% were squashed due to address aliasing in the signatures.

Radix shows a large commit overhead for TCC and SEQ. This is because, as we will see in Section 6.2, chunks in Radix use a large number of directory modules compared to other applications. Moreover, most of these modules record writes. Radix implements a parallel radix sort algorithm that ranks integers and writes them into separate buckets for each digit. The writes to these buckets are random depending on the integer, and have no spatial locality. This results in a large number of directory modules that record writes per chunk, and in serialization for non-overlapped protocols.

Previous work on Scalable TCC [6] shows smaller overheads for Radix and Barnes, but this was in the context of software-defined transactions. Such transactions are much larger than the automatically-generated chunks of this work. As a result, the transactions do not commit as frequently, leading to better scalability. However, they need software to define them.

Ocean, Cholesky, and Raytrace attain superlinear speedups. The reason is that the single-processor runs can only use a single L2 cache, while the parallel runs use 32 or 64 L2 caches.

6.2. Directories Accessed per Chunk Commit

Figures 9 and 10 show the average number of directory modules accessed per chunk commit in the ScalableBulk protocol. The data is shown for the SPLASH-2 and PARSEC applications. The figures show data for each application for 32 and 64 processors, and the average across applications. Each bar shows the number of directories that record at least one write (*Write Group*) and of those that record only reads (*Read Group*).

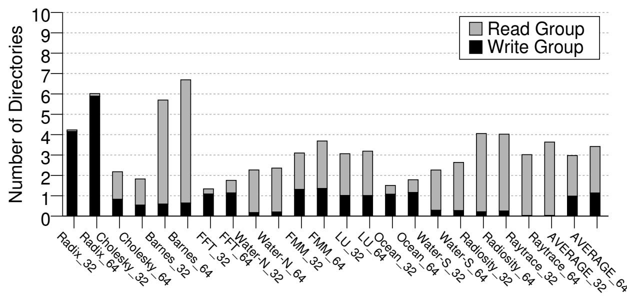


Figure 9. Number of directories accessed per chunk commit in SPLASH-2.

The scalability of distributed commit protocols such as ScalableBulk, TCC, and SEQ depends on chunks accessing a small number of directory modules. We see from the figures that most applications access an average of 2–6 directories per chunk commit. These numbers are significantly larger than the ones reported in the Scalable TCC paper [6]. Moreover, in some applications such as Barnes, Canneal, and Blackscholes, chunks access a much larger set of directories. Radix is especially challenging in that practically all of the directories in the group record writes.

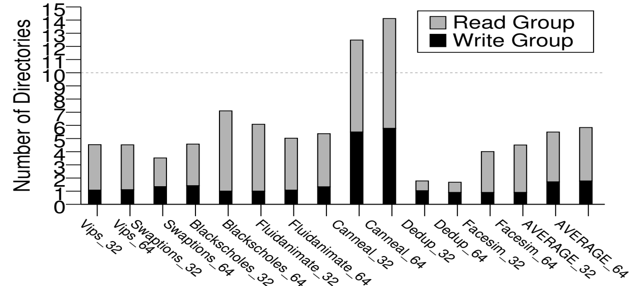


Figure 10. Number of directories accessed per chunk commit in PARSEC.

Large groups are especially harmful for TCC and SEQ, due to their inability to concurrently commit two overlapping chunks. However, ScalableBulk suffers no noticeable commit overhead thanks to its ability to overlap groups through the use of signatures.

Figures 11 and 12 show the distribution of the number of directory modules accessed per chunk commit in the ScalableBulk protocol for the SPLASH-2 and PARSEC applications. We can see that, in some applications, there is a significant tail of chunks with high numbers of directories accessed.

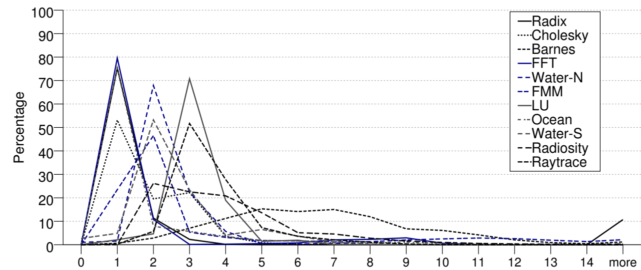


Figure 11. Distribution of number of directories accessed per chunk commit in SPLASH-2 for 64 processors.

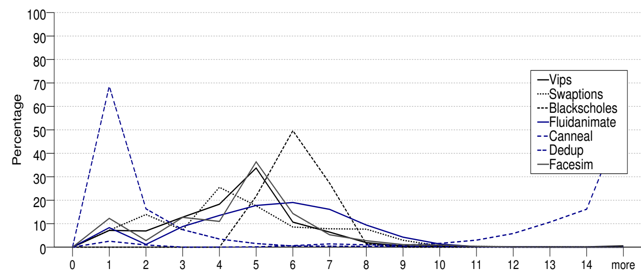


Figure 12. Distribution of number of directories accessed per chunk commit in PARSEC for 64 processors.

6.3. Chunk Commit Latency

Figure 13 shows the distribution of the latency of a chunk commit operation. The data corresponds to the average of all the applications running on 64 processors. The mean latencies for ScalableBulk, TCC, SEQ, and BulkSC are 91, 411, 153, and 2954 cycles respectively. For 32 processors, the mean latencies are 74, 402, 107, and 98 cycles, respectively. ScalableBulk not only has lower latencies than all of the existing protocols, but also scales well. BulkSC has the worst scaling behavior.

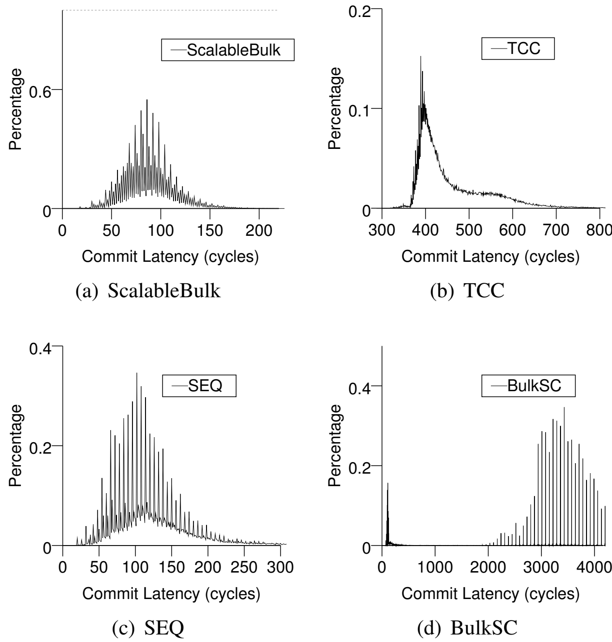


Figure 13. Distribution of the latency of a chunk commit operation.

6.4. Chunk Commit Serialization

In order to analyze the divergent commit latencies shown in Section 6.3, we measure two additional metrics: the bottleneck ratio and the chunk queue length.

6.4.1. Bottleneck Ratio

We define the Bottleneck Ratio as "the number of chunks in the process of forming groups" over "the number of chunks that have successfully formed groups and are in the process of completing the commit". We exclude from the numerator those chunks that are forming groups that will later be squashed. This ratio is sampled every time that a new group is formed. A high ratio signifies that groups are taking a long time to form, most likely due to a bottleneck — e.g., in the case of the less-overlapped protocols, group formation is stalled waiting for another group to commit. A low ratio signifies that groups are getting formed and processed quickly through the system.

A high bottleneck ratio does not necessarily imply a high commit overhead because the commit latency can be hidden by the execution of the next chunk in the processor. However, given an application, a high bottleneck ratio in a protocol compared to another protocol is a good indicator that group formation is taking a longer amount of time.

Figures 14 and 15 show the bottleneck ratios in SPLASH-2 and PARSEC for ScalableBulk, TCC and SEQ. We do not show data for BulkSC because it does not form groups.

From the figures, we see that the bottleneck ratio in ScalableBulk is uniformly low and is about 1 on average. This roughly means that chunks spend about the same amount of time forming a group as committing the group. In contrast, SEQ and especially TCC, have higher bottleneck ratios. Some of the applications that have a high bottleneck ratio are those with the most commit overhead in Figures 7 and 8, such as Radix, Barnes, FMM, Blackscholes, and Canneal.

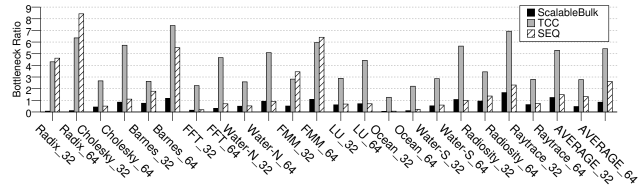


Figure 14. Bottleneck ratio for SPLASH-2 codes.

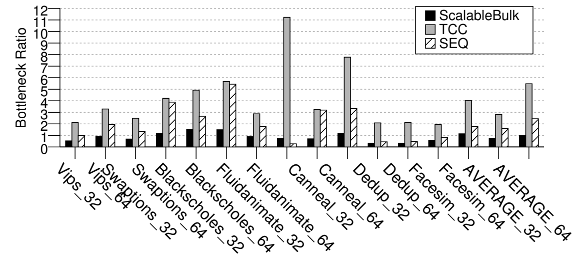


Figure 15. Bottleneck ratio for PARSEC codes.

6.4.2. Chunk Queue Length

The Chunk Queue Length is the number of chunks in the whole machine that are queued waiting to commit. A completed chunk gets queued in the TCC and SEQ protocols when the directory modules that it accessed overlap with those accessed by earlier, yet-uncommitted chunks. Chunks do not get queued in ScalableBulk because ScalableBulk enables full overlap of chunk commits using signatures. We sample the chunk queue length every time that a new group is formed. A long chunk queue means that a completed chunk has to wait for a long time to commit. Therefore, it signifies commit serialization.

Figures 16 and 17 show the average chunk queue lengths in TCC and SEQ for all the applications with 64 processors. Long chunk queues do not necessarily imply high commit overhead because the commit latency can be hidden by the execution of other chunks. However, we can see that applications such as Radix, Blackscholes, and Canneal, which have high commit overheads in Figures 7 and 8, do in fact have long chunk queues.

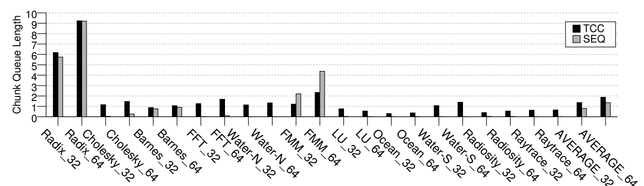


Figure 16. Chunk queue lengths in SPLASH-2 codes.

6.5. Traffic Characterization

Figures 18 and 19 show the number and distribution of the messages in the network for the different coherence protocols. The data is shown for each application running on 64 processors. For a given application, the bars are labeled as *S* (for ScalableBulk),

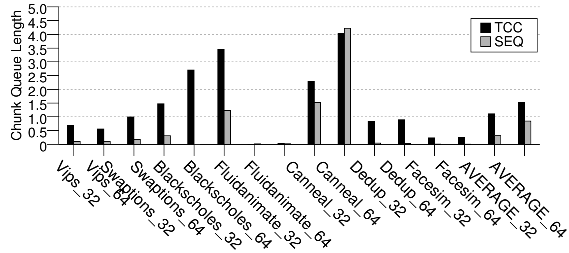


Figure 17. Chunk queue lengths in PARSEC codes.

T (for TCC), *Q* (for SEQ), and *B* (for BulkSC), and are normalized to the number of messages in TCC. The messages are classified into five classes: reads of a cache line from memory (*MemRd*), reads of a cache line from another cache, either in state shared (*RemoteShRd*) or in state dirty (*RemoteDirtyRd*), and two classes of commit-related messages (*LargeCMessage* and *SmallCMessage*).

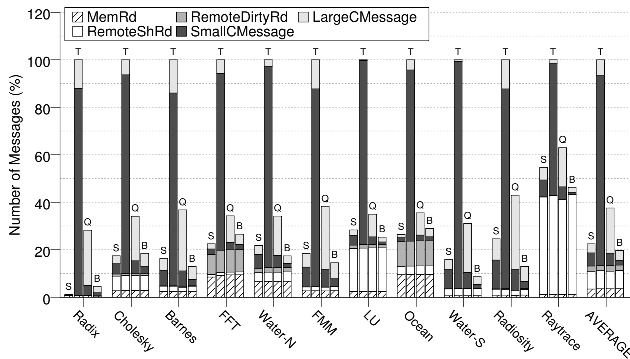


Figure 18. Message characterization in SPLASH-2. *S*, *T*, *Q*, and *B* stand for ScalableBulk, TCC, SEQ, and BulkSC, respectively.

LargeCMessage and *SmallCMessage* are large and small messages, respectively, in the commit protocol. For example, in ScalableBulk, the *LargeCMessage* are those that carry signatures, namely *commit_request* and *bulk_inv* in Table 1, while *SmallCMessage* are the rest of the messages in Table 1.

From the figures, we see that TCC generates significantly more messages than the other protocols, and that these messages are mostly commit-related small messages. These messages are mostly the skip and probe messages. This results in a more congested network, potentially increasing the commit overhead and delaying the read messages.

7. Conclusions

To boost programmability and performance, researchers have proposed architectures that continuously operate on chunks of instructions. These systems must support chunk operations efficiently. In particular, in lazy conflict-detection environments, they must provide scalable chunk commits. Unfortunately, current proposals very often limit the overlap of conflict-free chunk commits.

This paper presented ScalableBulk, a novel directory-based protocol that enables highly-overlapped, scalable chunk commits. ScalableBulk uses hardware address signatures to optimize chunk operations. It introduces three general hardware primitives for scalable commit: preventing access to a set of directory entries, grouping directory modules, and initiating commit optimistically. Our results with SPLASH-2 and PARSEC codes with up to 64 processors show that ScalableBulk enables highly-overlapped chunk commits and scalable performance. Unlike previously-proposed schemes, it removes practically all commit stalls.

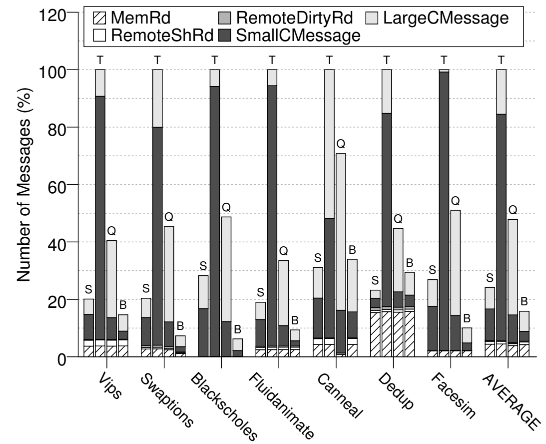


Figure 19. Message characterization in PARSEC. *S*, *T*, *Q*, and *B* stand for ScalableBulk, TCC, SEQ, and BulkSC, respectively.

Acknowledgments

We thank the anonymous reviewers and the students in the I-ACOMA group for their comments. Special thanks go to Reetuparna Das for allowing us to use her network simulator.

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 *Inter. Symp. on Microarchitecture*, December 2009.

[2] C. Blundell, M. M. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Inter. Symp. on Computer Architecture*, June 2009.

[3] 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 *Inter. Symp. on Computer architecture*, June 2007.

[4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Inter. Symp. on Computer Architecture*, June 2006.

[5] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Inter. Symp. on Computer Architecture*, June 2007.

[6] 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.

[7] 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 *Inter. Symp. on High Performance Computer Architecture*, February 2009.

[8] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2009.

Node Type	Successful Commit	Failed Commit: Collision Module is the Leader
Leader	R:commit_request → S:g → R:g → (S:commit_success & S:g_success (multicast) & S:bulk_inv (multicast)) → R:bulk_inv_ack (several) → S:commit_done (multicast)	R:commit_request → (S:g_failure (multicast) & S:commit_failure). Discard any later commit_recall. **or** R:commit_recall → R:commit_request → (S:g_failure (multicast) & S:commit_failure)
Non-Leader	(R:commit_request & R:g) → S:g → R:g_success → R:commit_done	R:commit_request & R:g_failure (from leader)

Table 4. Messages sent and received by a directory module.

Node Type	Failed Commit: Collision Module is not the Leader
Leader	R:commit_request → S:g → R:g_failure → S:commit_failure
Before Collision Module	(R:commit_request & R:g) → S:g → R:g_failure (from Collision Module)
Collision Module	(R:commit_request & R:g) → S:g_failure (multicast). Discard any later commit_recall. **or** (R:commit_request & R:commit_recall) → R:g → S:g_failure (multicast) **or** (R:g & R:commit_recall) → R:commit_request → S:g_failure (multicast)
After Collision Module	R:commit_request & R:g_failure

Table 5. Messages sent and received by a directory module.

[9] 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 *Inter. Symp. on Computer Architecture*, June 2004.

[10] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *Inter. Symp. on Computer Architecture*, June 2008.

[11] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Inter. Symp. on Computer Architecture*, June 2007.

[12] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Inter. Symp. on Computer Architecture*, June 2008.

[13] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Inter. Symp. on Computer Architecture*, June 2007.

[14] 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.

[15] 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.

[16] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6), 2002.

[17] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Inter. Symp. on Microarchitecture*, December 2009.

[18] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing kilo-instruction multiprocessors. In *International Conference on Pervasive Systems*, July 2005.

[19] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *Inter. Symp. on Computer Architecture*, June 2007.

[20] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Inter. Symp. on High Performance Computer Architecture*, February 2007.

Appendix A: Ordering of Messages in a Directory

In this appendix, we describe the ordering of the messages sent and received by a directory module, both in a successful commit and in a failed one. There are two subcases in a failed commit, namely that the Collision module is the leader of the failed group or that it is not. If it is not, then we need to consider the three types of modules in the failed group, as shown in Figure 20. The types of modules are: those before the Collision module, the Collision module, and those after the Collision module.

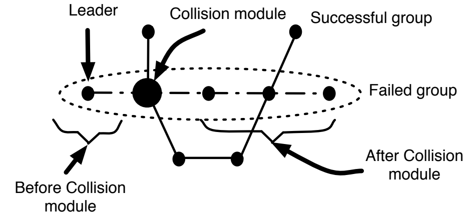


Figure 20. Types of directory modules in a failed group where the Collision module is not the leader.

Table 4 shows the succession of messages sent and received by a directory module in the case of (1) a successful commit and (2) a failed commit where the Collision module is the leader of the failed group. Table 5 shows the case of a failed commit where the Collision module is not the leader of the failed group. In the tables, *S:msg* means that a message *msg* is sent by the directory, while *R:msg* means that *msg* is received by the directory. Moreover, *ev1* → *ev2* indicates that event *ev1* precedes event *ev2*, while *ev1 & ev2* means that *ev1* and *ev2* can happen in any order (but both of them should happen). The message types are those presented in Table 1.

As an example, consider Table 5 for the Collision module. There are three possible message orderings. In one of them, the module receives a *commit_request* and a *g* (grab) message in any order. Then, since the module finds out that this group fails, it multicasts a *g_failure* message to all of the other group members. Later, if and when it receives a *commit_recall* message, it discards it. In the second case, the module receives a *commit_request* and a *commit_recall* message. At that point, it waits for the reception of a *g* message. After that, it multicasts a *g_failure* message. Finally, in the third case, the module receives a *g* and a *commit_recall* message. At that point, it waits for the reception of a *commit_request* message. After that, it multicasts a *g_failure* message. The other table entries can be described similarly.