BulkCompactor: Optimized Deterministic Execution via Conflict-Aware Commit of Atomic Blocks*

Yuelu Duan, Xing Zhou, Wonsun Ahn, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Recent proposals for determinism-enforcement architectures are able to honor the dependences between threads through a commit step that often becomes a performance bottleneck. As they commit code blocks (or *chunks*) in a round-robin order, if one chunk gets squashed due to a conflict, its successors also observe a stall. We call this effect *transitive squash delay*.

This paper proposes a novel, high-performance approach to deterministic execution based on *Conflict-Aware* commit. Rather than committing chunks in strict round-robin order, the idea is to skip those chunks with conflicts and deterministically execute them slightly later. The scheme, called BulkCompactor, largely eliminates transitive squash delay, "compacts" the chunk commits, and substantially speeds-up execution. With BulkCompactor, the squash overhead is $O(N)$ rather than $O(N^2)$ as in round-robin. We describe BulkCompactor designs for machines with centralized or distributed commit. Finally, a simulation-based evaluation shows that BulkCompactor delivers performance comparable to nondeterministic systems. For example, for 32 processors, BulkCompactor incurs an average execution overhead of 22% over a nondeterministic system. The round-robin scheme's average overhead is 133%.

1. Introduction

Current shared-memory systems are nondeterministic. Multiple runs of the same parallel application with the same input often execute with different interleavings and may even produce different results. The lack of determinism makes writing, testing, and debugging parallel code harder. For example, due to nondeterminism, a bug may be difficult to reproduce, test cases that provide enough coverage may be hard to develop, and correct code may be trickier to write in the first place.

To address this problem, there have recently been several proposals for architectures that enforce determinism in parallel execution (e.g. [3, 5, 11, 12, 14]). Using different combinations of hardware and software, these systems generally run code sections in parallel, without allowing the different cores or threads to communicate. Then, in a deterministic data merge step, they resolve the dependences between the sections.

A key difficulty of this approach is that, to honor the dependences between the different threads, the merging step can become a performance bottleneck. The Grace [5] and DMP-TM [11] schemes run the parallel section in a transactional manner, executing what we call an atomic block or *chunk*. Each core or thread buffers the state that the chunk generates in a private buffer. Then, in the merge step, these buffers are committed to memory in a deterministic *round-robin* manner. During the merging, we may find a data dependence between two chunks, also known as a *conflict*. This implies that one of them has used inconsistent state and, therefore, needs to be squashed and restarted.

Other proposals give up on maintaining the dependences across threads that were specified in the program. They perform a fast merging step that results in an unconventional memory consistency model [3, 12, 14]. The result can be unintuitive executions.

Round-robin commit is costly because chunks have to commit in a total pre-defined order. If a chunk gets squashed due to a conflict, it delays not only itself but very likely the commit of all of its successor chunks as well. We call this effect *transitive squash delay*. In addition, as multiple chunks get squashed, their transitive squash delay accumulates, imposing more delay on successor processors. Overall, we find that the total squash delay increases as $O(N^2)$, where N is the processor count. With many processors, the overhead can be high — e.g., the execution of `ocean-nc` from SPLASH-2 with 32 processors is 4x slower than on a nondeterministic system.

This paper makes the key observation that, since conflicts are deterministic, rather than committing chunks in a strict round-robin order, we can skip those chunks with conflicts and deterministically execute them slightly later. With this, we can largely eliminate transitive squash delay, "compact" the chunk commits, and substantially speed-up execution — all while retaining deterministic execution. We call this *Conflict-Aware* commit scheme *BulkCompactor*. With BulkCompactor, the squash delay increases as $O(N)$. We also propose a variation of BulkCompactor called BulkCompactor-S (for scalable) that is only concerned with conflicts between neighbor chunks. In both algorithms, starvation and unfairness are addressed with simple solutions. Overall, BulkCompactor delivers performance that is comparable to that of a nondeterministic system.

The contributions of the paper are:

• We propose BulkCompactor, a novel, high-performance scheme for deterministic execution based on *Conflict-Aware* commit. BulkCompactor temporarily postpones the commit of chunks with conflicts to avoid transitive squash delay. The overall squash delay is $O(N)$.

• We describe the design of BulkCompactor as extensions to a nondeterministic multiprocessor. We present two designs: one for a machine with a centralized commit, and the other for a machine with distributed commit.

• We evaluate BulkCompactor and the conventional round-robin scheme with detailed simulations using SPLASH-2 and Parsec applications. BulkCompactor delivers high-performance deterministic execution. For 32-processor executions, BulkCompactor incurs an average execution overhead of 22% over a nondeterministic

* This work was supported in part by the National Science Foundation under grant CCF-1012759 and by Intel under the Illinois-Intel Parallelism Center (I2PC).

system. The round-robin scheme's average execution overhead is 133%.

This paper is organized as follows. Section 2 reviews background material; Section 3 presents BulkCompactor and BulkCompactor-S; Section 4 shows their design; Section 5 evaluates them; Section 6 lists related work; Section 7 concludes; and Appendix A presents an overhead analysis.

2. Background

2.1. Deterministic-Execution Architectures

Proposed architectures that enforce deterministic execution (e.g., [3, 5, 11, 12, 14]) generally run parallel applications in a succession of two steps. In the first step, processors run code sections independently, without communicating; in the second step, they execute a deterministic data merge step where they resolve the inter-processor (or inter-thread) dependences in the previous sections.

Of these architectures, we are interested in those like Grace [5] and DMP-TM [11] that honor the dependences between threads specified in the program and support sequential consistency — since their execution is always intuitive. In these systems, the parallel section runs in a *transactional* manner, executing an atomic block or *chunk*. Each core or thread buffers the state that the chunk generates in a private buffer, and can only read from memory or from that buffer. Then, in the merge step, these buffers are committed to or merged with memory in a deterministic total order.

During the merge, the system knows which memory locations were read or written by each chunk. It checks for data dependences between chunks, also known as *conflicts*. On a dependence, a chunk may have read stale data or overwritten good data, and so it must be squashed. Specifically, if a chunk that wrote to line *X* commits, then the system cannot commit a chunk that read or wrote line *X* — since the second chunk may have read stale data or incorrectly overwritten data generated by the first chunk. Squashing a chunk involves discarding the state it generated and returning the thread's execution to the beginning of the chunk.

The conflict detection in these systems is *lazy*. It means that the trigger for the system to check for conflicts is when a chunk commits — rather than when the chunk performs an access.

In these architectures, the merging step can easily become a performance bottleneck, due to frequent chunk squashes. A major reason for this is that these architectures use *round-robin* to totally order the commit of the chunks from the different processors: $P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_{N-1} \rightarrow P_0 \rightarrow \cdots$. We discuss it next.

2.2. Overheads with Round-Robin Merge

Lazy, round-robin merge suffers the performance overheads shown in Figure 1. The figure shows three processors executing chunks with a round-robin $P_0 \rightarrow P_1 \rightarrow P_2$ commit policy. The unlabeled arrows are the commit token passing. When chunk C_0 in P_0 commits, the system detects a dependence with chunk C_1 in P_1. C_1 is then squashed and re-executed.

An obvious overhead is the squashed work of C_1, which slows down P_1. It is shown as *(1)* in the figure. The overall squashed work is highly related to the application's runtime behavior and number of processors. Applications with higher inter-thread communication typically have a higher squash rate. Also, a chunk's squash rate often increases with the number of processors.

In a squash, the use of round-robin commit induces an additional performance overhead compared to a platform where chunks

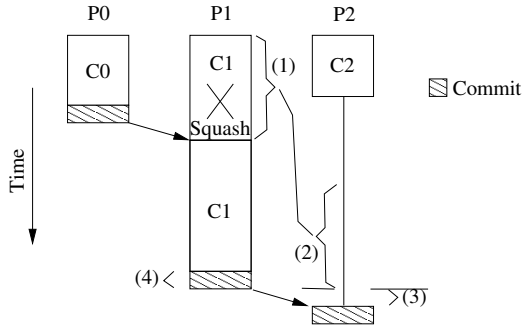


Figure 1. Performance overheads.

are allowed to commit in any order — e.g., a nondeterministic one. Specifically, when a chunk gets squashed, part (or all) of the squashed execution may be visible as stall to the successor processors. This transmitted stall is shown as *(2)* in the figure. In this case, *(1)* is visible in its entirety to P_2. Moreover, all of the successor processors can potentially be affected by it as well. This transmitted stall accumulates and hence tends to increase with the number of processors.

Token passing is another overhead, shown as *(3)* in the figure. Since we require a total order of chunks, we experience delay to transfer the token between processors. This overhead is higher the more physically distributed the machine is.

Another overhead is the part of the commit operation that appears in the critical path of the processor, shown in the figure as *(4)*. Finally, there is conflict detection overhead, not shown in Figure 1.

All these overheads are mostly orthogonal to determinism except for the transmitted stall (the second one), which appears in round-robin deterministic systems. Our paper focuses on largely eliminating this performance overhead.

2.3. Bulk Architecture for Chunk Execution

To compare a non-deterministic environment to a deterministic one with either round-robin commit or the improved commit we propose in this paper, we use as a common substrate the chunk-based Bulk architecture [26]. This is a cache-coherent architecture where processors continuously execute chunks. A chunk is a dynamically-formed group of contiguous instructions. A chunk executes atomically and in isolation. As it executes, it buffers the state it generates in the L1 cache. In addition, hardware Bloom filters encode all the addresses read and written by the chunk into a read (R) and a write (W) signature. Conflict detection between concurrent chunks is performed lazily, when a chunk commits. At that point, the committing chunk's W signature is intersected against the other chunk's R and W signatures. If both intersections are null, there is no conflict; otherwise, the second chunk is squashed.

We consider two Bulk designs: one with a centralized commit arbiter (BulkSC [10]) and one with distributed directory modules and no arbiter (ScalableBulk [24]). Both are nondeterministic.

In BulkSC, when a chunk completes execution, it sends out its signatures (R_i, W_i) to the commit arbiter. The arbiter uses the signature pair to decide whether to commit the chunk. The arbiter maintains a committing queue, which contains the W signatures of all of the currently-committing chunks. If the intersection of R_i and W_i with any of the W in the committing queue is not null, there is a conflict and the processor is denied the commit request. Otherwise, the chunk is permitted to commit, and the arbiter informs the

requester, which de-allocates R_i and W_i. In addition, the arbiter inserts W_i into the committing queue, updates the directory (if there is one), and sends W_i to relevant processors to check for conflicts and possibly squash their chunks. The committing queue is used by the directory to reject loads and stores whose address overlaps with the W of the committing chunks.

In ScalableBulk, the directory is partitioned into multiple on-chip directory modules based on address ranges. Each directory module is associated with a node and there is no commit arbiter. When a chunk completes execution, it sends out its signatures to all of the directory modules that may have addresses present in the signatures. This group of directory modules then coordinate to determine whether the chunk can commit. It can commit only if the addresses that the chunk accessed do not overlap with those of another chunk that is currently committing. Two groups of directory modules can proceed with concurrent commits even if they include common directory modules.

3. Conflict-Aware Commit of Chunks

In this section, we further analyze the limitation of the state-of-the-art deterministic commit and then propose to solve the problem with conflict-aware commit.

3.1. Transitive Squash Delay in Round-Robin

As indicated above, state-of-the-art deterministic systems commit chunks in a round-robin order. In these systems, when a processor squashes a chunk, it potentially slows down all of the chunk's successor processors. This is repeated in Figure 2, which is simplified to assume equal chunk sizes, no commit cost and no token passing overhead. In the figure, there is a conflict between chunk C_0 and C_1, and between C_2 and C_3. Since we use a lazy scheme, after processor P_0 commits C_0, C_1 gets squashed due to the conflict. Thus, when P_1 gets the commit token, it cannot pass it immediately to P_2 because it has to re-execute C_1. Thereby, P_2 ends up waiting the equivalent of a chunk's execution, even though its chunk does not conflict with any of the predecessor chunks. In effect, the re-execution overhead of C_1 in P_1 is *transmitted* to P_2 and all of the successor processors. We call this transmitted delay the *Transitive Squash Delay*.

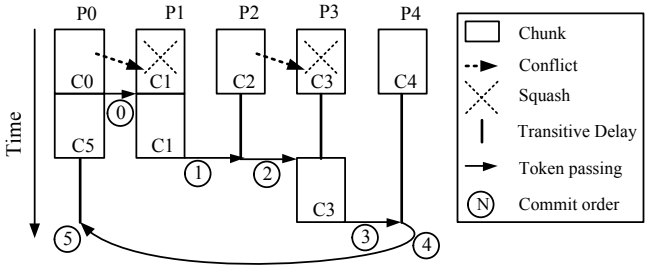


Figure 2. Round-robin deterministic chunk commit.

Transitive squash delays accumulate. As illustrated in Figure 2, P_4 suffers the sum of the delays introduced by the squashes in P_1 and in P_3 (squashed by P_2). The transitive delay generated by all the processors in one *Round* of commits (i.e., one commit per processor in the machine) ends up accumulating in the last processor in the commit order. With more processors, the overall transitive delay observed in a round is likely to grow, because (1) the squash rate per processor increases given that more concurrent chunks are running, and (2) there are more processors that can get squashed and contribute to the transitive delay in the last processor. With some reasonable assumptions, it can be shown that both factors grow linearly with the number of processors. Consequently, the overall squash delay increases as $O(N^2)$, where N is the number of processors (Appendix A.2).

As will be shown in our evaluation, such quadratically-growing overhead is modest for 4-8 processors. However, it can become substantial with more processors. For 32 processors, the slowdown of round-robin over a nondeterministic system is 2.3x on average for all the applications, and 4x for `ocean-nc`.

Meanwhile, in a nondeterministic chunk system, the squash delay grows as $O(N)$ (Appendix A.1). Our goal is to reduce the overhead of a deterministic system to reach the same level.

3.2. BulkCompactor: Minimizing Squash Delay

BulkCompactor is a new deterministic execution scheme that focuses on minimizing the transitive squash delay. The idea is that a chunk that does not conflict with earlier chunks does not need to be delayed. In addition, chunks that conflict with earlier chunks are deterministically *postponed* to the next round to retry the commit, instead of following the round-robin order. Later, in the next round, every processor continues to commit chunks as usual — as long as they are conflict-free.

Figure 3 illustrates how BulkCompactor works. The figure shows the same example as Figure 2 extended into three rounds of commit. The token is still passed in round-robin order. In the first round, P_0 commits C_0 and then squashes C_1. Instead of waiting for C_1's re-execution, the token *bypasses* P_1 and is sent to P_2. P_2 commits C_2 while C_1 is *postponed* to the next round. By doing so, P_2 avoids the transitive delay of C_1's squashed work. Similarly, C_3 is squashed by C_2 and also postponed. After C_4 commits, the token is passed back to P_0 to start a new round. In this new round, P_0, P_2 and P_4 can commit new chunks, while P_1 and P_3 commit the re-executed chunks — assuming there are not cross-chunk conflicts. In the third round, each processor commits its next chunk because there are no conflicts. Overall, BulkCompactor sequentially commits conflict-free chunks in each round and, on a conflict, postpones the destination chunk to the next round for retry. The result is a highly-compacted execution of chunks. It produces a different chunk commit order than the round-robin one but, since dependences are deterministic, this order is also deterministic across executions of the same application and input.

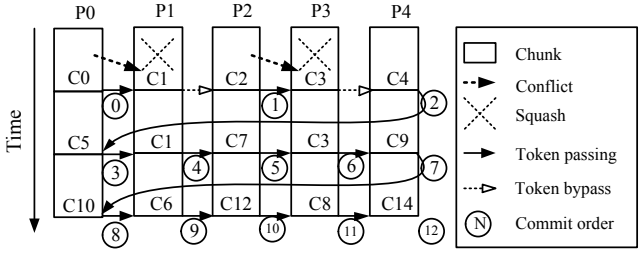


Figure 3. BulkCompactor deterministic chunk commit.

BulkCompactor eliminates transitive squash delay in a round. If a chunk is squashed, rather than delaying successor chunks, it re-executes but its commit gets postponed to the next round. Chunks non-conflicting with their predecessor chunks in the same round commit without waiting for such re-executions.

Interestingly, these features are quite similar to nondeterministic chunk-based systems such as TCC [13] and BulkSC [10]. In these systems, if a chunk is squashed, it is re-executed and retried for commit later, without affecting other processors; furthermore, conflict-free chunks can always commit regardless of the behavior of other processors. Our analysis shows that the overall squash delay of BulkCompactor follows $O(N)$ (Appendix A.3).

In summary, the key novelty of BulkCompactor is to compact the schedule of chunk commits. The schedule is determined at runtime, depending on the conflict behavior of the application.

3.3. Enforcing Deterministic Postponement

To understand BulkCompactor's operation, Figure 4 shows a model of processors executing chunks. The execution is divided into *Rounds*, which contain one chunk per processor. Within a round, a deterministically-picked *Leader* is the first chunk to commit. The commit token originates in the leader and is passed in a fixed order around the chunks in the round. Based on the order of token visit, a chunk has *Predecessor* and *Successor* processors (and chunks) in this round. When the commit token arrives at a chunk, the chunk will be committed or postponed. The commit token travels with the *Non-Postponed Set*. This set includes the chunks in this round that have already committed (and, therefore, are not postponed). This set is gradually expanded as the commit token travels within the round. The chunk that receives the commit token is called the *Commit-Attempting* chunk.

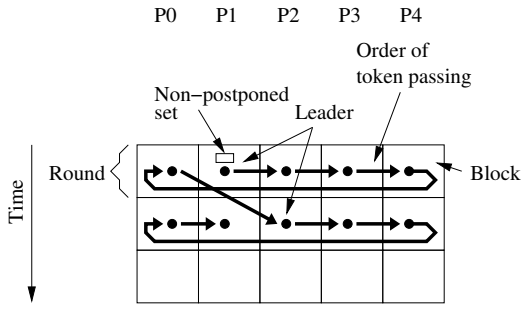


Figure 4. Model of BulkCompactor's execution.

With BulkCompactor, it is crucial that the decision of whether to postpone the commit of a chunk to the next round be deterministic. If it is, it easily follows that the rest of the system is deterministic as in the case of round-robin. While there can be many decision algorithms, we propose two simple rules that ensure determinism:

Rule 1. A chunk C_i is always postponed if it has a conflict with any previously committed chunk C_j in the same round, where $0 \leq j < i$.

Rule 2. A chunk C_i is never postponed because of a conflict with a chunk from an earlier round.

When the commit token reaches a chunk, BulkCompactor needs to decide whether the chunk should be committed or postponed. If the chunk has already been squashed by a predecessor chunk in the same round, then the chunk is automatically declared postponed. Otherwise, the chunk needs to finish its execution. Only then can BulkCompactor, with the full information of the addresses accessed by the commit-attempting chunk, determine whether or not there is a conflict with any predecessor chunk. BulkCompactor determines this by comparing the addresses accessed by the commit-attempting chunk to those accessed by the chunks in the non-postponed set. If there is a conflict, BulkCompactor postpones the chunk as per Rule 1 — even though the conflict did not actually trigger the squash of the chunk in this run. Otherwise, the commit-attempting chunk commits and is added to the non-postponed set. After a round finishes and the token is passed to the leader of the next round, the non-postponed set is cleared as per Rule 2.

At all times, chunk squashes are only triggered by the conventional lazy-scheme protocol: as a chunk completes and commits, the protocol squashes all of the currently uncommitted chunks that have prematurely accessed a conflicting location. However, the processes of squashing and postponing are *decoupled*. This is because, while squashing is nondeterministic (a cross-chunk dependence may or may not cause the squash of the consumer chunk depending on the time when the consumer issues the consumer access), postponing must be made deterministic.

For instance, as Figure 5(a) shows, chunk C_1 may or may not be squashed by C_0, depending on whether the conflicting access in C_1 (*ld B*) is performed before C_0's commit completes. In the example, C_1 is not squashed. However, to keep execution deterministic, C_1 is always postponed to the next round (Figure 5(b)).

As indicated above, when a chunk is squashed by a predecessor chunk in the same round, the chunk is automatically postponed. However, the situation is different when a chunk is squashed by a chunk in an earlier round. In this case, as per Rule 2, the chunk simply restarts and will still be committed in the present round. This event can cause the only instance of transitive squash delay. For example, consider the case where the last chunk of a round has a conflict with the first one of the next round. As the earlier chunk commits, it will squash the second one. The latter will still restart and commit in the next round, potentially passing some transitive delay to its successors.

Figure 5(c) shows an example of this case, which also illustrates the nondeterminicity of the squashes. Chunk C_1 of P_1 is in the first round and C_2 of P_0 is in the second round, and they have a conflict. C_1 is executed and commits in the first round. At that point, depending on whether the consumer access in C_2 (*ld A*) has been performed or not, C_2 is squashed. In the example, it is squashed. However, as C_2 re-executes, it will commit in the second round (Figure 5(d)). This event may add some transitive squash delay to the chunks in the second round.

To summarize, the postponement detection enforces the two rules and guarantees a deterministic commit order of chunks regardless of the nondeterministic behavior of chunk squashes. As the commit order defines the thread interleaving, it follows that the execution of BulkCompactor is deterministic.

3.4. BulkCompactor-S: Scalable Postponement

BulkCompactor performs *full-round* postponement detection, in that a commit-attempting chunk compares its addresses to those of all of its predecessor chunks in the round that have not been postponed. This approach ensures that the transitive squash delay is minimized, since any conflict discovered triggers the postponement of the commit-attempting chunk. However, with many processors, the comparison operation becomes costly. The reason is that, as the conflict rate is usually low, the non-postponed predecessor chunks (collected in the non-postponed set) often are *all* of the predecessor chunks in the round. Hence, a commit-attempting chunk C_i often has to be compared to all of its predecessor chunks, which consumes energy and either is slow or requires substantial hardware. The total number of operations required for postponement

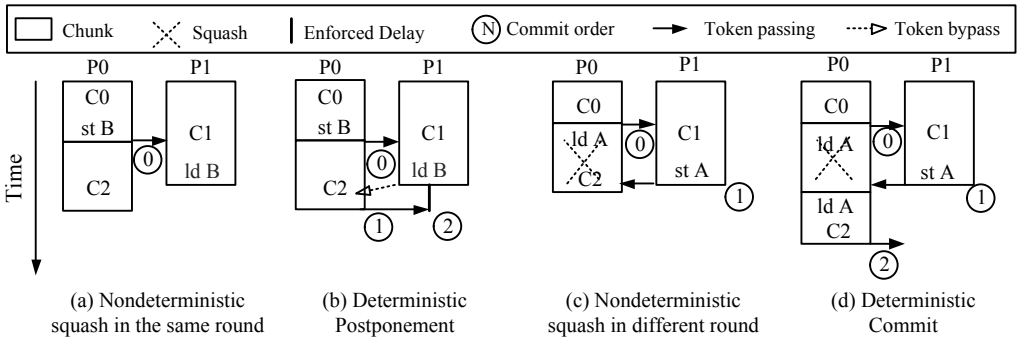


Figure 5. While squashing is nondeterministic, postponement is deterministic.

detection is in the order of $O(N^2)$ for each round, where N is the processor count.

To reduce the number of comparisons, we propose BulkCompactor-S (for *Scalable* BulkCompactor). BulkCompactor-S performs *partial-round* postponement detection. Specifically, BulkCompactor-S decides whether to postpone a commit-attempting chunk C_i by only considering conflicts with at most its M immediate predecessor chunks. M is a small constant called *Range* such that $1 \leq M \ll N$. Consequently, on a squash, the chunk's commit will be postponed only if the squash is induced by one of the chunk's M immediate predecessors. In addition, BulkCompactor-S gradually removes entries from the non-postponed set as the commit token travels between chunks. At any time, the non-postponed set has at most M entries.

The advantage of BulkCompactor-S is that it reduces the number of comparisons between chunks. Its shortcoming is that it may be unable to eliminate some of the transitive squash delay that BulkCompactor eliminates. Specifically, if a chunk C_i has a conflict with a predecessor that is earlier than C_{i-M}, then C_i will not be postponed and any delay resulting from chunk C_i's squash may be visible to i's successors.

Fortunately, two reasons help minimize the delay propagated to chunk C_i's successors. The first one is that, since the conflict occurs between relatively far-off chunks, it is typically detected far before the commit token reaches C_i. Hence, if C_i got squashed, by the time the token reaches C_i, C_i has likely had the time to finish most of its re-execution, and the delay propagated will be small. The second reason is that experimental evidence suggests that programs have a certain *conflict locality*, meaning that a thread tends to conflict with neighboring threads more often than with far-off threads — especially in nearest-neighbor algorithms. As a result, by not checking for far-off dependences to decide on postponements, chunks are unlikely to suffer much transitive squash delay. Overall, BulkCompactor-S with small M can discover and avoid most of the transitive squash delay.

Compared to BulkCompactor, BulkCompactor-S enforces these adjusted determinism rules:

Rule 1'. A chunk C_i is always postponed if it has a conflict with any previously committed chunk C_j in the same round where $MAX\{i-M, 0\} \leq j < i$.

Rule 2'. A chunk C_i is never postponed because of a conflict with a chunk from an earlier round or with a chunk C_j in the same round where $0 \leq j < MAX\{i-M, 0\}$.

Figure 6 shows an example of BulkCompactor-S with *M = 2*. There is a conflict between chunks C_0 and C_1, and between C_0 and C_3. C_1 is squashed and postponed. Hence, its squash delay is not transmitted. C_3 is squashed by C_0. However, C_3 is conflict-free with the current non-postponed set — composed of only C_2 because C_1 is postponed. Consequently, C_3 is not postponed and its re-execution causes transitive delay to C_4. As suggested by the example, with BulkCompactor-S, the transitive squash delay is likely to be kept low (e.g., C_1 was postponed but not C_3), which means that BulkCompactor-S may not be much slower than BulkCompactor. Meanwhile, partial-round postponement detection needs fewer comparison operations for each round than in BulkCompactor. Specifically, the number of operations is in the order of $O(N \times M) = O(N)$ rather than $O(N^2)$ as in BulkCompactor. Hence, BulkCompactor-S consumes less energy doing the comparisons. In addition, the number of comparisons per chunk is reduced from $O(N)$ to $O(M)$, which means that BulkCompactor-S needs less storage for the non-postponed set, and either needs less comparison hardware or compares faster.

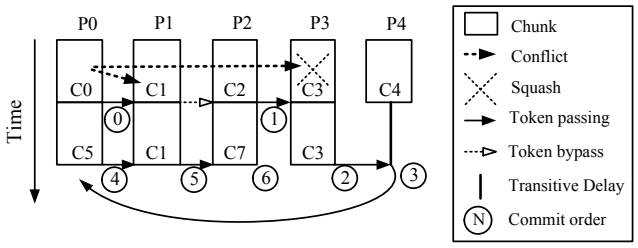


Figure 6. BulkCompactor-S operation with $M = 2$.

The squash delay of BulkCompactor-S is in the order of $O((N-M)^2 + M)$ (Appendix A.4). Consequently, with a small range M, the delay would be similar to round-robin's. In practice, however, we will see in Section 5 that, for 32 processors, BulkCompactor-S with $M = 4$ or $M = 8$ attains almost the same performance as BulkCompactor. This low overhead is due to the two reasons mentioned before. Thus, BulkCompactor-S with a small M avoids most of the transitive squash delay.

3.5. Handling Starvation and Fairness

A simple implementation of BulkCompactor and BulkCompactor-S may induce starvation. It occurs when a chunk continuously fails to commit due to a predecessor that repeatedly conflicts with the chunk and postpones it. For instance, consider two processors P_0 and P_1, where P_0 has higher priority to commit than P_1 in every round. If P_0 continuously commits chunks that conflict with the chunk that P_1 is trying to commit (C_1), then C_1 will be postponed indefinitely. To solve this issue, both BulkCompactor and BulkCompactor-S shift the leader processor at every round (Figure 4). This simple solution ensures

that each processor can commit at least one chunk every N rounds, thus avoiding starvation and guaranteeing progress.

Unfortunately, the support described does not ensure fairness. In the worst case, a processor might only commit one chunk for every N rounds, while other processors make fast progress. This may happen when a processor's chunk is squashed every time except when the processor becomes the leader. With the insight that the leader processor is guaranteed to commit a chunk, we set the leader to be the one that suffers the most postponements. With this solution, fairness is attained.

4. Design

We design BulkCompactor and BulkCompactor-S as extensions to the Bulk architecture [26] (Section 2.3). The components of the design include constructing a deterministic postponement detector (Section 4.1), ensuring deterministic chunk building (Section 4.2), and enforcing deterministic conflict detection (Section 4.3). Supporting BulkCompactor and BulkCompactor-S in other chunk-based platforms such as TCC [13] or the software-based Grace system [5] may involve somewhat different issues. In the following, we describe the three components.

4.1. Deterministic Postponement Detector

The *Postponement Detector* (or *p-detector* for short) is a hardware module that ensures deterministic chunk postponement and commit order. It uses address signatures for its operation. In the following, we outline a design for a centralized system like BulkSC [10] and one for a distributed system like ScalableBulk [24]. The designs correspond to BulkCompactor; those for BulkCompactor-S require only small modifications.

4.1.1. Centralized Design

In a design with a centralized arbiter like BulkSC, the p-detector is placed together with the arbiter. The commit token and the non-postponed set are kept in the p-detector rather than circulated between the processors. The non-postponed set is an array with as many entries as the number of processors. The entries for the processors that have successfully committed a chunk in the current round contain the W signature of the chunk; the entries for those that have postponed the chunk in the current round contain a marker. The commit token is an index into the non-postponed set array, pointing to the processor that holds the token.

Processors inform the p-detector of two events: (1) when they complete a chunk and request to commit it, and (2) when their chunk gets squashed (in this case, the message includes which processor squashed it). With this information, the p-detector is able to deterministically postpone and commit chunks. When it decides to commit a chunk, it informs the arbiter, which will proceed as in BulkSC — namely, send confirmation to the requesting processor (which will then start a new chunk) and proceed with the directory update. We now describe each of the two processor-to-p-detector messages.

When a processor completes a chunk, it sends its R and W signatures to the arbiter/p-detector module and stalls. Unlike in BulkSC, the arbiter never rejects any of these requests. The p-detector buffers the signatures. Later, when the p-detector's commit token reaches this processor number, the p-detector checks if the chunk has a conflict with any of its non-postponed predecessors in the same round. For this, it intersects the chunk's R and W signatures with the W signatures of all its non-postponed predecessors in the same round, which are stored in the non-postponed set array. If any intersection is not null, there is a conflict. In this case, the p-detector silently postpones the executed chunk to the next round, records this fact in the non-postponed set array, and retains the R and W signatures. Otherwise, the p-detector stores the chunk's W signature in the non-postponed set array and requests the arbiter to commit the chunk. In either case, the p-detector then moves the commit token to the next entry.

When a processor suffers a chunk squash, it informs the p-detector and restarts the chunk. The message includes the ID of the processor that caused the squash, which came from the invalidation message that triggered the squash. Since the p-detector keeps at all times a table with which round each processor is currently executing in, the p-detector knows if the squashing processor is in the same round as the squashed processor. If it is, the p-detector marks the squashed chunk as postponed; otherwise, it takes no action. In either case, the p-detector discards the squashed chunk's R and W signatures (if it had them).

When the p-detector points the commit token to a chunk that is to be postponed, it simply moves the token to the next entry. When the p-detector completes a round, it clears the non-postponed set array. It then moves to commit the leader of the next round. At any time, if the p-detector points the commit token to a chunk for which it neither has the R and W signatures (because the chunk has not completed yet) nor has it recorded as postponed for this round, it waits. Finally, a processor stalls from the time it requests a chunk commit until it gets a commit confirmation (at which point it starts a new chunk) or it gets the chunk squashed (at which point it re-starts the chunk).

Figure 7(a) outlines the main structures of the p-detector and arbiter. The W signatures in the non-postponed set (center) are intersected with the R and W signatures of the incoming chunks (left), resulting in a commit or a postponement. Committing chunks store their W signatures in the non-postponed set and in the arbiter (right), preventing overlapping loads and stores from accessing the directory.

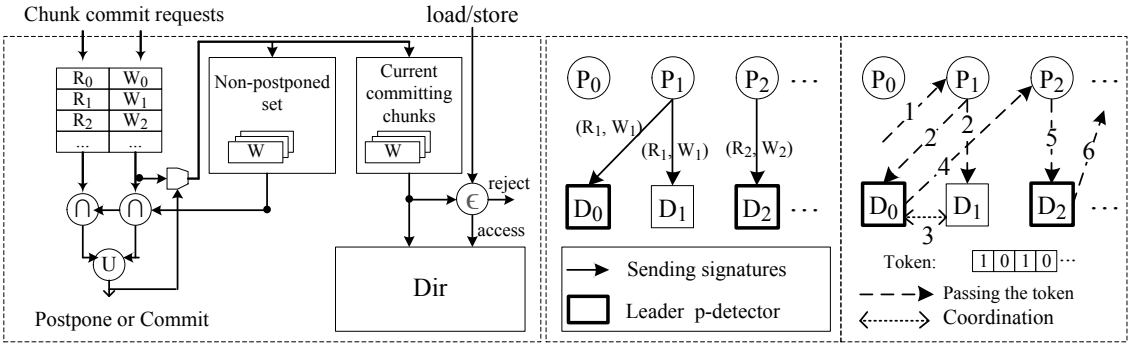
4.1.2. Distributed Design

ScalableBulk modifies the BulkSC protocol to work in a machine with a directory distributed across multiple modules, each handling a range of memory addresses. We now enhance such a design by associating a p-detector with each directory module.

A key idea of ScalableBulk is that a processor only communicates with the subset of directory modules that are home to the data read or written by the chunk. We call these the *Relevant* directories/p-detectors. Consequently, in the distributed BulkCompactor, a processor informs its relevant p-detectors (and only them) when it wants to commit a chunk or when its chunk gets squashed. Only these p-detectors will store the chunk's R and W signatures when the chunk requests to commit, and the chunk's W signature when the chunk is part of the non-postponed set.

Another ScalableBulk feature is that in a group of relevant directories, there is a *Leader* that coordinates the actions of the *Group* and communicates with the requesting processor. Such leader is the lowest-numbered directory in the group. In the distributed BulkCompactor, we use such a concept.

In the distributed BulkCompactor, passing the commit token involves sending protocol messages between processors and p-



(a) Centralized P-Detector (b) Distributed P-Detector

Figure 7. P-detector for deterministic chunk postponement and commit order.

detectors. Specifically, when a processor has the token for a chunk, it sends it to its relevant p-detectors. These p-detectors, under the coordination of its leader, decide whether the chunk should be committed or postponed. The actual coordination is much simpler than in ScalableBulk, since only one group of p-detectors is active at a time — unlike in ScalableBulk, where multiple groups are active concurrently. After the decision is made, the leader sends the token to the next processor in the commit order, which forwards it to its relevant p-detectors.

BulkCompactor needs a structure to keep which processors have committed their chunks in the current round so far. This information is carried in the commit token. Specifically, the token is a bit vector with as many bits as the number of processors. For the processors that have committed their chunks so far in the current round, the bit is set; for those whose chunks have been postponed or not considered yet, the bit is clear.

The actual distributed BulkCompactor algorithm is as follows. When the relevant p-detectors for a chunk C_i receive the commit token, they first check if the chunk has been squashed by another one in the current round. If so, the chunk is postponed. Otherwise, the p-detectors intersect the R_i and W_i signatures of the chunk against the W_k signatures of all the predecessor chunks already committed in this round. Such predecessors have their bit set in the commit token. In addition, their W_k signatures are stored locally if the local p-detector was relevant to those chunks. If the local p-detector does not have a given W_k signature, it means that such a chunk did not access addresses local to this directory module and, therefore, the intersection would be null anyway. As the intersections are performed, the leader aggregates the information from all the group members. If the result is that all the intersections are null, the chunk can commit. At this point, the leader sets the bit in the commit token and informs all the p-detectors/directories in the group, which save the W_i signature and proceed to commit the chunk as in ScalableBulk. If, instead, at least one intersection is not null, there was a conflict and the chunk is postponed. In all cases, the leader then sends the token to the next processor to commit.

Figure 7(b) shows an example of the algorithm. After P_1 finishes a chunk, it sends the R and W signatures to its relevant p-detectors, with a chosen leader D_0. In the meantime, P_2 also completes a chunk and sends out the signatures. Once P_1 receives the commit token from an earlier processor, it forwards the token to all of its relevant p-detectors (message *2* in Figure 7(b)), which perform signature intersection. After the leader D_0 collects the results (message *3*), it decides to postpone the chunk, hence leaving P_1's bit in the token clear, and forwards the token to P_2 (message *4*). P_2 forwards the token to D_2, which decides to commit the chunk. D_2 sets P_2's bit in the token, initiates the commit, and passes the token to the next processor.

Overall, this is a distributed algorithm with minimal amount of data communicated. Still, the latency of the token-passing messages appears in the critical path of chunk commit or postponement. To prevent such latency from becoming too high in large machines, it may be beneficial to provide fast interconnection links for token passing.

4.2. Deterministic Chunk Building

A requirement of a chunk-based deterministic machine is that chunks be built deterministically. BulkCompactor accomplishes this by terminating a chunk when it reaches a certain maximum size and by handling certain events that could introduce nondeterminism. We discuss the latter. Recall that BulkCompactor is concerned with application-level determinism only.

4.2.1. Cache Overflow

The overflow of data generated by an uncommitted chunk from the cache (or speculative buffer) is typically nondeterministic. It depends on the machine state. When overflow is about to happen, the chunk cannot continue and is usually forced to commit its partial state. This would be a non-deterministic chunk termination.

In the round-robin scheme, nondeterminism can be avoided. Before any part of the overflowing chunk C is allowed to commit, all of the previous chunks are committed. Then, the section of C before the overflow is committed. Finally, the rest of C is executed and committed before passing the commit token.

Unfortunately, this solution fails in BulkCompactor or BulkCompactor-S. We cannot commit *any* section of C before completing and checking the full chunk. The reason is that the rest of C could contain a conflict with a prior chunk and require C to be postponed.

To solve this issue, we allocate a buffer in main memory that temporarily buffers data generated by an uncommitted chunk that overflows from the cache. When the chunk finally completes and commits, the data is copied from the buffer back to non-speculative space. We call the buffer *sufficient private buffer* or *S-buffer*. The solution is similar to the buffer in the Unbounded Transactional Memory (UTM) system [1] and the unlimited-sized overflow area of Prvulovic *et al.* [23].

The buffer is hash-indexed with the address for fast access. Moreover, thanks to BulkCompactor using signatures, it can be manipulated efficiently. Specifically, the S-buffer only needs to be accessed on cache misses whose address is found in the W signature. In addition, read-only data that is evicted from the cache does not

Processors & Interconnect	Memory Subsystem	Application	Input Size
Cores: 2-issue in-order at 1GHz	Private D-L1:	barnes	4096 nbody
4 to 32 cores with centralized p-detector	Size/assoc/line:	fmm	4096 parts
64 cores with distributed p-detectors	64KB/4-way/64B	lu-nc	512x512 matrix
Maximum chunk size: 2,000 instructions	Latency: 1 cycle	ocean-nc	258x258 ocean
Interconnect: 2D bidirectional torus	Shared distributed L2:	radiosity	default
Interconnect hop latency: 7 cycles	Size/assoc/line:	radix	262144 keys
Signature size: 2K bits	4MB/8-way/64B	raytrace	teapot
Signature organization: Like in BulkSC [10]	Latency to local module: 5 cycles	bodytrack	simmedium
Range for BulkCompactor-S: 4 and 8	S-buffer & main memory:	fluidanimate	simmedium
Simultaneous chunks per core: 2	Avg. latency: 150 cycles	streamcluster	simmedium
Page size: 256KB	S-buffer size: 64KB	swaptions	simmedium
Default architecture: Centralized p-detector		x264	simmedium

Table 1. Architecture and application parameters.

need to be stored in the S-buffer because the R signature keeps a record of the addresses read. Finally, since BulkCompactor builds chunks with a fixed maximum size, the S-buffer can be statically sized so that all the overflowed data fits in it. This is unlike past schemes like UTM's or Prvulovic *et al.*'s buffers.

In our evaluation, we use a cache line replacement algorithm that follows LRU but tries avoid replacing speculatively-written cache lines. Our evaluation shows that the S-buffer is rarely accessed and thus imposes negligible performance overhead.

4.2.2. Other Events

Other events could also introduce nondeterminism. These events include system calls, interrupts, program inputs, and program exceptions. We briefly discuss how to handle them to ensure determinism. First, system calls cause the termination of the currently running chunk because BulkCompactor is concerned with application-level determinism rather than system-level. Since they are part of the executable, they terminate the chunks deterministically. However, care must be taken that nondeterministic operating system behavior does not affect the state used by the application. Second, typical interrupts such as timer interrupts can be delayed to the end of the currently running chunk. They are executed at chunk boundaries and, therefore, do not affect chunks. Third, program inputs need to be deterministically provided in every run of the application for the execution to be deterministic. Finally, program exceptions such as division by zero can be either reproduced deterministically or eliminated if they are an artifact of the speculative execution. Specifically, the chunk with the exception needs to be re-executed after all of its predecessor chunks commit, to ensure a deterministic architectural state.

4.3. Deterministic Conflict Detection

We adopt signatures to perform conflict detection [9]. Signature operations can introduce false positive conflicts, namely two sets of addresses that are disjoint, when represented as signatures, may appear to overlap with each other. Ordinarily, this would pose no major problem because such false positives are also deterministic. Unfortunately, a problem occurs if we use physical addresses — which change from run to run — to generate signatures. While true conflicts will be deterministic, false positive conflicts will not. To solve this problem, BulkCompactor uses virtual addresses to construct signatures. Such addresses are identical (or can be made to be identical) across program runs. In addition, the signature encoding mechanism is assumed to be identical across machines.

Another problem is that, in a distributed machine like the one described in Section 4.1.2, addresses are distributed across the different directories based on their physical values. Therefore, unless we do something, the same data structures (hence, the same virtual addresses) will likely be mapped to different directories in different runs. As a result, when we intersect virtual-address signatures in a directory/p-detector to decide on chunk postponement and commit, we will intersect different addresses across runs. While true conflicts will be the same, false positive conflicts will vary across runs, making the algorithm non-deterministic.

To solve this problem, we must assign the same virtual addresses to the same directory modules across runs. We do this by choosing a large page size and using some *page offset* bits (which are the same for virtual and physical addresses) to map physical space to different directories. For example, if we have 16 directories, we use the four most-significant bits of the page offset to assign space to the different directories — i.e., one directory gets all of the 0000 offsets, another all of the 0001, etc. In this way, a given virtual page, irrespective of what physical page it maps to in this particular run, will be split into the different directories in a deterministic manner. As a result, the virtual-address signatures in each directory will be identical across runs. This scheme works best with large pages. Hence, we use 256-Kbyte pages.

4.4. Applying Existing Optimizations

BulkCompactor deterministic systems are compatible with and can benefit from many performance optimizations that have been proposed for chunk-based systems. For example, supporting multiple in-progress chunks per thread [10] is useful to overlap commit latency with next-chunk execution. As another example, data forwarding between two dependent chunks as in the case of Thread-Level Speculation can be applied to reduce the number of squashes.

5. Evaluation

5.1. Evaluation Setup

We use the Intel PIN [16] infrastructure to model the Bulk multicore architecture [26] running the default nondeterministic environment (used as baseline) plus extensions to support the BulkCompactor, BulkCompactor-S, and round-robin deterministic schemes. We model both the centralized and distributed designs of Sections 4.1.1 and 4.1.2. We model all the components, including chunk building, chunk squash and roll-back, and deterministic postponement and commit. We perform experiments for 4–64 processors. We run 12 applications from SPLASH-2 and Parsec to completion. The cache hierarchy is sized for the modest working set sizes of the applications. The parameters of the architecture and applications are shown in Table 1.

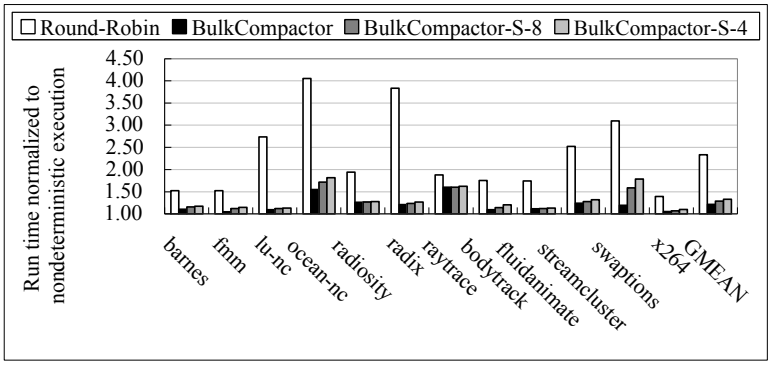


Figure 8. Execution time on different architectures for 32 processors. BulkCompactor-S uses ranges of 4 or 8. The bars are normalized to the nondeterministic architecture.

5.2. Overall Performance

Figure 8 compares the execution time of the applications running on several architectures for 32 processors. The figure shows bars for the round-robin, BulkCompactor and BulkCompactor-S schemes (the latter with ranges of 4 and 8) normalized to the nondeterministic architecture. Based on the geometric mean, BulkCompactor only incurs a 22% execution overhead over the nondeterministic platform. The round-robin scheme's execution overhead is 133%. Hence, execution on BulkCompactor takes slightly over half the time it takes in round-robin. BulkCompactor-S-4 and BulkCompactor-S-8 incur an overhead of 34% and 29%, respectively. These overheads are moderately higher than BulkCompactor's because BulkCompactor-S suffers from some transitive delays as the range decreases.

It can be shown that the execution overheads of our round-robin scheme are comparable to those of DMP-TM [11] for 4, 8, and 16 processors, which are the data points reported for DMP-TM.

The `raytrace` code is unusual in that BulkCompactor incurs an overhead similar to round-robin. A common pattern in `raytrace` is that a chunk often conflicts with the chunks of many other processors. This happens because `raytrace` uses critical sections that conflict frequently. As a result, when the first chunk commits, it squashes most of the other concurrent chunks. Hence, BulkCompactor cannot compact the execution. This suggests that a code's synchronization pattern can have a noticeable impact on the behavior of BulkCompactor. This issue is a subject for future work.

Interestingly, for a large fraction of the applications, BulkCompactor-S-4 and BulkCompactor-S-8 perform almost like BulkCompactor. The reason is that, in these programs, the work is divided into processors in a way that, if a conflict appears, a processor tends to conflict mostly with its neighboring processors. On the other hand, for some programs such as `ocean-nc`, `bodytrack` and `swaption`, the conflicts are more distributed. Therefore, BulkCompactor-S suffers some transitive squash delays. The characterization section gives more details.

Figure 9 shows an example of the scalability of the execution time overhead as the number of processors increases. The figure corresponds to `fmm` for 4–32 processors. With increased number of processors, round-robin's overhead grows significantly, whereas BulkCompactor's and BulkCompactor-S's only grow moderately. This is the case for most of our applications.

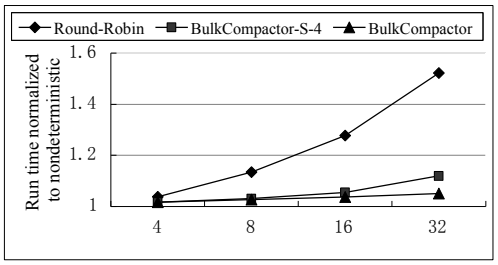


Figure 9. Scalability of the execution time overhead for `fmm`.

5.3. Sensitivity to the Chunk Size

Figure 10 compares the relative overheads of the round-robin and BulkCompactor schemes for different maximum chunk sizes. For a given maximum chunk size, we measured the execution time increase of round-robin over the nondeterministic system, and of BulkCompactor over the nondeterministic system. Then, we plot them normalized to the first one. The figure corresponds to `barnes` with 32 processors. For very large chunk sizes such as 100K instructions, BulkCompactor's overhead is similar to round-robin's. The reason is that, with such a chunk size, the conflict rate is very high and therefore the execution is severely serialized in both schemes. For smaller chunk sizes, the two schemes differ. Therefore, relative to round-robin, BulkCompactor is most attractive for moderate chunk sizes such as 2–8K instructions.

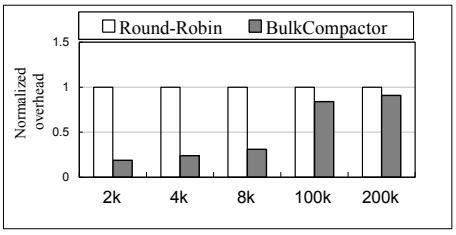


Figure 10. Comparing round-robin to BulkCompactor with different maximum chunk sizes for `barnes`.

5.4. Sensitivity to the Token-Passing Latency

We now focus on the BulkCompactor architecture with distributed p-detectors (Section 4.1.2). In this architecture, the commit token is passed explicitly between processors, potentially adding

Apps	Transitive Ratio			Bypass Rate (%)		Conflict Degree	Conflict Distance	S-Buffer (for BC)		
	RR	BC	BC-S	BC	BC-S	BC	BC	Chunks Store in S-Buffer	Dirty Lines Displaced	Displaced Lines per Chunk
barnes	9.6	1.1	2.5	11.3	5.3	7.1	9.2	0.01	0.00	34.5
fmm	3.8	1.0	1.8	3.2	2.5	5.3	7.1	0.02	0.00	2.1
lu-nc	8.4	1.1	1.3	17.5	10.6	3.4	4.7	1.57	0.01	1.7
ocean-nc	10.3	1.2	2.6	18.2	7.4	3.6	7.3	0.00	0.00	2.8
radiosity	5.7	1.1	1.1	29.5	15.5	2.8	3.6	0.00	0.00	21.0
radix	21.2	1.1	1.2	17.7	13.6	3.1	2.5	9.37	0.02	1.3
raytrace	6.8	1.4	6.5	84.3	80.6	16.6	3.3	0.00	0.00	3.6
bodytrack	3.1	1.0	1.5	12.6	7.8	2.4	6.3	0.00	0.00	2.2
fluidanimate	2.9	1.1	1.2	23.3	20.6	2.2	4.0	0.00	0.00	0.0
streamcluster	12.4	1.1	2.5	14.3	10.5	3.3	5.9	0.00	0.00	1.8
swaptions	8.2	1.1	4.2	11.3	8.7	2.3	12.3	0.00	0.00	0.0
x264	3.1	1.0	1.3	7.1	5.0	6.7	7.5	0.97	0.00	2.1
Average	8.0	1.1	2.3	20.9	15.7	4.9	5.9	1.00	0.00	6.1

Table 2. Characterizing deterministic schemes: round-robin (RR), BulkCompactor (BC), and BulkCompactor-S with M=8 (BC-S).

stall to the application. To assess this effect, we measure the average time it takes for the token to travel a round trip (i.e., leaving from a processor, visiting all the other processors, and returning to the first one). We include the time needed for the p-detectors to coordinate with each other, but not the chunk commit time or any stall time. We consider network-hop latencies of 2–7 cycles in a 64-processor machine. Figure 11 shows the round-trip latency normalized to the average time it takes for a chunk to execute and send its signatures to its relevant directories. If the ratio is higher than one, a processor will finish the execution of its chunk and stall before it gets the commit token again.

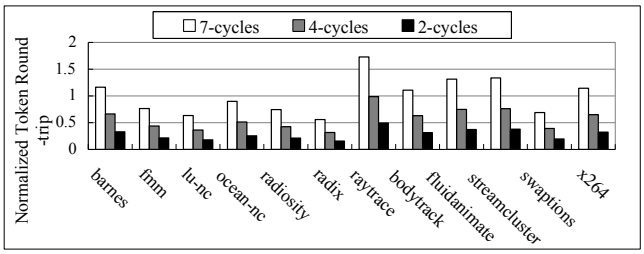


Figure 11. Commit-token round-trip time for different network-hop latencies normalized to the chunk execution time.

The figure shows that, for the default 7-cycle hop time, several applications have a ratio higher than 1. Hence, the token-passing operation becomes a bottleneck for some applications. With smaller hop latencies, this effect disappears. For example, for 2 cycles, the token passing-latency is hidden behind chunk execution. Hence, supporting low-latency token passing is very necessary for distributed architectures.

5.5. General Characterization

To gain insight into the deterministic schemes, Table 2 characterizes round-robin (RR), BulkCompactor (BC), and BulkCompactor-S with a range of 8 (BC-S) for 32 processors and the centralized p-detector architecture. In the following, we call inherent squash delay the time wasted by a processor when one of its own chunks gets squashed or postponed. Based on this, Column *Transitive Ratio* shows the ratio of (inherent squash delay plus transitive squash delay) over the inherent squash delay. Generally, a higher ratio leads to a higher execution overhead. On average, RR has a ratio of 8.0, while BC has a ratio of only 1.1 because it eliminates most of the transitive squash delay, and BC-S has a ratio of 2.3. The *Bypass Rate* column shows the fraction of times that a token bypasses a chunk and, therefore, the chunk is postponed. A moderate value is usually good because it implies overhead reduction against RR, as transitive squash delay is avoided. On average, BC has a bypass rate of 21% while BC-S has a rate of 16%. The next two columns show data only for chunks that conflict with other chunks in the same round. Column *Conflict Degree* shows the average number of chunks they conflict with, while Column *Conflict Distance* shows the average processor distance of the conflict. (We set the conflict distance between two conflicting chunks C_i and C_j to be $|i - j|$). We see that, save for a few exceptions, both values tend to be modest. On average, they are 4.9 and 5.9, respectively. The latter explains why BC-S does well.

The *S-Buffer* columns show the behavior of the buffer described in Section 4.2.1. The *Chunks Store in S-Buffer* column shows the number of chunks that need to use the S-Buffer per 1,000 chunks. It is about 0.0 for most applications, and the average is only 1.0. The *Dirty Lines Displaced* is the average number of dirty cache lines displaced into the S-Buffer for every 1,000 L1 accesses. This number is even lower than the previous column. Finally, the *Displaced Lines per Chunk* column shows the average number of dirty cache lines displaced into the S-Buffer per chunk that needs to use the S-Buffer. This value is 6.1 on average. These values indicate that the S-Buffer is sparsely accessed and imposes minor performance overhead.

6. Related Work

There has been significant interest on deterministic execution from different layers of the computing stack. The DMP-TM scheme proposed as a part of the DMP project [11] and the Grace system [5] are the most related works to ours. Both of them enforce a pre-defined, round-robin commit order of chunks, thus incurring a quadratic squash delay growth as we show in our analysis. The DMP-O scheme [11] uses ownership tracking to enforce memory access ordering. In case of ownership violation, it can introduce significant serialization and resulting performance degradation. The DMP-B scheme proposed in CoreDet [3] and further exploited in RCDC [12] employs relaxed consistency to reduce the serialization of DMP-O. Calvin [14] proposes a pure hardware implementation similar to the DMP-B scheme. Our scheme, instead, enforces sequential consistency.

Kendo [21] is a runtime library that imposes ordering restrictions for synchronization primitives and guarantees determinism for

race-free programs. The dOS [4] operating system enforces determinism for process groups by applying DMP-O at page granularity, thus inherently incurring similar inefficiencies as DMP-O. The Determinator [2] is a deterministic operating system that relies on a deterministic micro-kernel and explicit message passing to emulate shared-memory access. The Deterministic Parallel Java project [6] proposes a type and effect language that prohibits nondeterministic communication at the language level, enforced by compiler verification. Stream-based programming languages such as StreamIt [25] provide determinism as the deterministic message channel handles all the communication. Finally, some works check if a program behaves deterministically in a nondeterministic execution environment (e.g., [7, 8, 20]).

A related problem is that of execution recording and deterministic replay. The goal is not to enforce a deterministic thread interleaving like BulkCompactor, but to deterministically replay a given initial interleaving. The existing proposals include both hardware-level solutions (e.g., [15, 17, 19]) and software approaches (e.g., [18, 22, 27]). Much of the effort has focused on generating a minimal execution log, to save space. This has lead to schemes where log entries correspond to the execution of large code structures similar to chunks [15, 17]. The speed of the execution replay has been less critical, unlike in the deterministic execution schemes.

7. Conclusion

This paper proposed a novel, high-performance approach to deterministic execution based on *Conflict-Aware* commit. The goal was to eliminate the commit bottleneck that proposed determinism-enforcement architectures suffer as they honor the dependences between threads. They use round-robin commit, which induces transitive squash delay.

Our key observation was that, since conflicts are deterministic, rather than committing chunks in a strict round-robin order, we can skip those chunks with conflicts and deterministically execute them slightly later. The result is that transitive squash delays are mostly eliminated, chunk commits are compacted, and execution time is substantially reduced — all while retaining deterministic execution. With our scheme, called BulkCompactor, the overall squash delay increases as $O(N)$, rather than as $O(N^2)$ as in round-robin.

We described the design of BulkCompactor as extensions to the Bulk chunk-based nondeterministic multiprocessor. We presented two designs: one for a machine with a centralized commit point, and the other for a machine with distributed commit. We evaluated BulkCompactor and the round-robin scheme with detailed simulations. The results showed that BulkCompactor delivers deterministic execution at a performance that is comparable to a nondeterministic chunk-based system. For example, for 32-processor executions, BulkCompactor only incurred an average execution overhead of 22% over the nondeterministic system. The round-robin scheme's average execution overhead was 133%.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, February 2005.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-enforced Deterministic Parallelism. In *OSDI*, October 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, March 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, October 2010.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, October 2009.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, October 2009.

[7] J. Burnim and K. Sen. Asserting and Checking Determinism for Multithreaded Programs. *CACM*, June 2010.

[8] J. Burnim and K. Sen. DETERMIN: Inferring Likely Deterministic Specifications of Multithreaded Programs. In *ICSE*, May 2010.

[9] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, June 2006.

[10] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, June 2007.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, March 2009.

[12] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, March 2011.

[13] L. Hammond, P. G. Gyarmati, C. Kozyrakis, K. Olukotun, B. D. Carlstrom, B. Hertzberg, V. Wong, M. Chen, J. D. Davis, M. K. Prabhu, and H. Wijaya. Transactional Memory Coherence and Consistency (TCC). In *ISCA*, June 2004.

[14] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, February 2011.

[15] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, June 2008.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, June 2005.

[17] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, June 2008.

[18] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, March 2009.

[19] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies using Strata. In *ASPLOS*, October 2006.

[20] A. Nistor, D. Marinov, and J. Torrellas. InstantCheck: Checking the Determinism of Parallel Programs Using On-the-Fly Incremental Hashing. In *MICRO*, December 2010.

[21] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, March 2009.

[22] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, October 2009.

[23] M. Prvulovic, M. J. Garzaran, L. Rauchwerger, and J. Torrellas. Removing Architectural Bottlenecks to the Scalability of Speculative Parallelization. In *ISCA*, June 2001.
[24] X. Qian, W. Ahn, and J. Torrellas. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *MICRO*, December 2010.
[25] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, March 2002.
[26] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore Architecture for Improved Programmability. *CACM*, December 2009.
[27] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, March 2011.

A. Squash Delay Analysis

A.1. Nondeterministic Chunk Systems

For simplicity, we make the following assumptions for all the chunk-based systems compared:
1. On a chunk squash, a fixed L cycles of work are lost.
2. The conflict rate between two concurrent chunks is C for all pairs of chunks.
3. $0 < C \ll 1$ and $0 < C \times N \ll 1$, where N is the number of processors in the machine.

Given two concurrent chunks with a conflict rate C, the worst case is for one of them to get squashed by the other one at every conflict. In this case, the *squash rate* or probability of that chunk getting squashed is $S_2 = C$. With k concurrent chunks (where $2 \leq k \leq N$), the worst-case squash rate for a chunk is $S_k = 1-(1-C)^{k-1}$. Since C has a very small value, $S_k \approx C \times (k-1)$. For SPLASH-2 and Parsec applications, $S_{64} \approx 0.05$ [24] when the chunk size is $\approx$ 2,000 instructions. With a squash rate of S, the squash delay is $L \times S$.

In a nondeterministic chunk execution system, when a chunk is squashed, its delay does not transmit. Hence, the total delay is proportional to the squash rate and chunk length:

$$O(DELAY_{Nondeterministic}(N)) = O(L \times S_N) = O(N).$$

A.2. Round-Robin Systems

In the round-robin scheme, an earlier chunk is never squashed by a later chunk. Thus, chunk C_k gets squashed every time it has a conflict with its predecessor chunks, namely C_0, C_1, ... C_{k-1}. Using the formula above, C_k has a squash rate of $S_{k+1} = C \times k$.

Consider now the last processor in the group, P_{N-1}. The second chunk in the group (C_1) gets squashed by conflicts with chunk C_0, and transitively causes P_{N-1} to delay with a probability of S_2. The third chunk in the group (C_2) gets squashed by conflicts with C_0 and C_1, and transitively causes P_{N-1} to delay with a probability of S_3. Thus the overall delay of P_{N-1} is the sum of the above cases:

$$O(DELAY_{RoundRobin}(N)) = O(L \times \sum_{k=2}^{N} S_k) = O(N^2).$$

A.3. BulkCompactor Systems

In BulkCompactor, any chunk with a conflict is simply postponed to the next round and does not cause any transitive squash delay. Consider the last chunk in the round, which is the one getting postponed when it has a conflict with any other chunk in the round. The probability of getting postponed is the probability of having a conflict with any other chunk in the round, namely, $S_N = C \times (N-1)$. Like in the nondeterministic systems, the total delay is proportional to the conflict rate of this chunk and chunk length:

$$O(DELAY_{BulkCompactor}(N)) = O(L \times S_N) = O(N).$$

A.4. BulkCompactor-S Systems

In BulkCompactor-S with range M, a chunk suffers transitive delays like in the round-robin scheme when it conflicts with another chunk that is farther than M. For a given processor P_k, the probability of such a conflict is S_{k+1-M}. The overall delay of P_{N-1} is obtained by considering all such probabilities from processor P_{M+1} to processor P_{N-1} and adding up all of the resulting delays:

$$O(DELAY_{out\_range}(N)) = O(L \times \sum_{k=2+M}^{N} S_{k-M}) = O((N-M)^2).$$

In addition, conflicts with chunks inside the M range cause postponements, and result in overall delays like in BulkCompactor, which are:

$$O(DELAY_{in\_range}(N)) = O(M).$$

Therefore, the overall squash delay is:

$$O(DELAY_{BulkCompactor-S}(N)) = O((N-M)^2 + M).$$

With small M, it is approximately $O(N^2)$.

If, due to conflict locality, all of the conflicts are between neighbor chunks within the range M, then the overall squash delay of both BulkCompactor and BulkCompactor-S is of the order of $O(M)$. However, the round-robin scheme does not scale as well and is $O(N \times M)$.