BulkSMT: Designing SMT Processors for Atomic-Block Execution[*]

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

Abstract

Multiprocessor architectures that continuously execute atomic blocks (or *chunks*) of instructions can improve performance and software productivity. However, all of the prior proposals for such architectures assume single-context cores as building blocks — rather than the widely-used Simultaneous Multithreading (SMT) cores. As a result, they are underutilizing hardware resources.

This paper presents the first SMT design that supports continuous chunked (or transactional) execution of its contexts. Our design, called *BulkSMT*, can be used either in a single-core processor or in a multicore of SMTs. We present a set of BulkSMT configurations with different cost and performance. We also describe the architectural primitives that enable chunked execution in an SMT core and in a multicore of SMTs. Our results, based on simulations of SPLASH-2 and PARSEC codes, show that BulkSMT supports chunked execution cost-effectively. In a 4-core multicore with eager chunked execution, BulkSMT reduces the execution time of the applications by an average of 26% compared to running on single-context cores. In a single core, the average reduction is 32%.

1. Introduction

There has been much interest recently in a class of shared-memory architectures where processors continuously execute atomic blocks of instructions — often called *chunks* or transactions. Broadly speaking, these architectures include research proposals such as TCC [9], Bulk [20], Implicit Transactions [23], ASO [24], InvisiFence [3], DMP [8] and SRC [16] among others. Their chunked execution mode can improve performance and software productivity. For example, it supports transactional memory [9, 16], high performance under strict memory-consistency models [3, 20, 24], deterministic execution [8], parallel program replay [13], and atomicity-violation debugging [12].

All of the proposals for such architectures have implicitly used as their building blocks single-context cores — rather than Simultaneous Multithreading (SMT) cores [21]. This is unfortunate, given that SMT cores are widely deployed [10, 11] and would likely be used in a commercial implementation of these architectures. It is therefore important to understand how SMT cores would support chunked operation, both individually and integrated into a multicore of SMTs.

SMT processors are attractive for chunked execution for some of the same reasons as they are interesting for conventional execution. Specifically, they enable a better utilization of the hardware resources in a core. Moreover, they support fast communication between contexts, which improves performance and minimizes energy consumption. However, they are also attractive for chunked execution in their own right. Indeed, by minimizing the cost of interaction between the multiple contexts of the same core, they can enable more aggressive, higher-concurrency forms of chunked execution — e.g., concurrent execution of *dependent* chunks.

On the other hand, the fact that SMT threads share caches and other hardware structures makes it intrinsically more difficult to support the execution of atomic, isolated chunks.

Given this state of the art, this paper contributes with the first SMT design that supports chunked (or transactional) execution of its contexts. We call it *BulkSMT*, and can be used either in a single-core processor or in a multicore of SMTs. In this paper, we first perform an analysis of the design space, and propose three BulkSMT configurations with different cost and performance: *squash* on conflict, *stall* on conflict, and *order* on conflict. Then, we describe a set of novel architectural primitives that enable chunked execution in an SMT core. Finally, we show how to augment the resulting SMT core to work in a multicore of SMTs that executes chunks.

We evaluate our BulkSMT designs using simulations of SPLASH-2 and PARSEC codes. Our results show that BulkSMT supports chunked execution cost-effectively — both when it runs in a single-core processor and when it is integrated in a multicore of SMTs. In a 4-core multicore with eager chunked execution, BulkSMT reduces the execution time of the applications by an average of 26% compared to running on single-context cores. In a single-core machine, the average reduction is 32%.

The paper is organized as follows: Section 2 provides a background; Section 3 shows the BulkSMT core design; Section 4 examines BulkSMT multicores; Section 5 discusses implementation issues; Sections 6 and 7 evaluate our designs; Section 8 discusses related work; and Section 9 concludes.

2. Background

2.1. Continuous Execution of Chunks

In blocked (or chunked) execution, a core continuously executes atomic blocks of instructions, also called transactions or chunks. There are several recent proposals of architectures that operate or can operate in this mode (e.g. [3, 8, 9, 16, 20, 23, 24]). These architectures have interesting capabilities related to performance and parallel software productivity.

In these systems, before a chunk starts, the processor hardware takes a register checkpoint. Then, as the chunk executes, the architecture records the addresses read and written by the chunk, and prevents the written data from being irreversibly merged with the memory system before the chunk is proven safe to commit. In most designs, no other processor is allowed to observe the intermediate state of the chunk as it executes. Consequently, the ar-

[*] This work was supported by the NSF under CCF-1012759, Intel under the Illinois-Intel Parallelism Center (I2PC), Spanish Gov. & European ERDF under TIN2007-66423, TIN2010-21291-C02-01 and TIN2007-60625, Aragon Gov. & European ESF under gaZ: T48 research group, Consolider CSD2007-00050, and HiPEAC-2 NoE (European FP7/ICT 217068). B. Sahelices is with Universidad de Valladolid, Spain (benja@infor.uva.es).

chitecture watches for data conflicts (i.e., RAW, WAW, and WAR dependences) between concurrently-executing chunks. If a conflict is found, typically one of the chunks is squashed and restarted. Squashing involves discarding the data updated by the chunk in the cache or buffer, and restoring the register checkpoint.

Another reason for squashes is the overflow of the cache or buffer that keeps the updates of the chunk (or a log of the values prior to such updates). When a chunk is squashed by an event that re-appears on re-execution (e.g., cache overflow), execution transfers to a special version of the code that guarantees forward progress.

A system with chunked execution needs to perform version management, conflict detection, and conflict resolution. Each of these operations can be performed eagerly or lazily. Version management deals with the storage of speculative and non-speculative data. The lazy policy buffers the speculative data in special storage, separate from the shared memory, until the chunk commits; the eager one stores the speculative data in place in the shared memory, and saves the prior values in a special buffer or log. Conflict detection refers to the detection of inter-thread conflicts. The eager policy detects them as soon as a chunk tries to perform the conflicting memory accesses; the lazy policy checks for conflicts when a chunk is ready to commit. Finally, conflict resolution refers to the action taken to deal with the conflict. The eager policy takes the action as soon as the eager conflict detector has detected the conflict; the lazy one takes the action when a chunk is ready to commit.

2.2. An Opportunity for Chunked Execution

Chunked-execution multiprocessors can attain higher performance if they can withstand data conflicts between concurrently-executing chunks without squashing. Recently, there have been several proposals for such systems. For example, some proposals use the Conflict Serialization model from database systems [2, 18]. The idea is to record and manage the conflicts between chunks as they execute, and then ensure that the chunks commit in the correct order. In practice, as we discuss in Section 8, supporting this additional level of concurrency has resulted in a complicated cache coherence protocol as in DATM [18] or in non-trivial bookeeping requirements to ensure correct ordering as in SONTM [2].

Other proposals involve a "state correcting" step. Specifically, in RetCon [4], execution proceeds after a data conflict occurs. However, when the chunk is ready to commit, RetCon attempts to fix its state by obtaining the current value of the variables that were involved in the conflict. Alternately, in transaction value prediction [15], a chunk uses a value that it predicts it will need in a future conflict.

In general, many of these schemes involve significant conceptual and hardware complexity — much of it resulting from the distributed nature of the multiprocessor hardware involved. In contrast, an SMT processor contains multiple hardware contexts that are in close proximity and share hardware structures such as caches and buffers. If such contexts run chunks that conflict with each other, the hardware can efficiently and quickly detect the conflicts, record them, and manage them, in order to attain some concurrency between the chunks.

Remarkably, none of the proposals for chunked-execution architectures has used SMT cores as its building block. There is, therefore, an opportunity to leverage SMT to support higher levels of chunk concurrency with simpler hardware than in the past. Uncovering such opportunity is the goal of this paper.

3. Chunked-Execution SMT Processors

Given an off-the-shelf SMT processor with L1 and L2 caches, we want to extend it to support chunked execution. In this section, we examine the design space and the basic hardware mechanisms required. The processor can be part of a multicore, although we ignore multicore effects until later sections.

3.1. Design Space

The extensions needed in the SMT processor to support chunked execution depend on our choices for the operations of Section 2.1. The preferred design points are shown in Table 1. For version management, it is simpler for the hardware to adopt an eager policy. This means allowing a speculative write from one context to update the cache and be immediately visible to the other contexts of the SMT processor. There is no need to save the old value of the written variable in a log, as long as speculative data is prevented from spilling from L2; we can always get the old value of the data from main memory. If L2 is about to overflow, the chunk is squashed. The alternative, lazy version management, would require separately buffering the state that each context is generating. For conflict detection, it is natural to do it eagerly between contexts, as soon as a conflicting access executes. Similarly, for conflict resolution, it is simpler for the hardware to do it eagerly, rather than keeping state and resolving the conflict at commit time.

Version Managmt.	Eager, but without updating main memory with speculative data. No log is needed
Conflict Detection	Eager
Conflict Resolution	Eager squash, eager stall, or eager order

Table 1. Preferred design points.

We propose three distinct eager conflict resolution schemes, as shown in Figure 1. They generate three very different BulkSMT design points. Consider a data dependence between two concurrent chunks as in Figure 1(a). In *SQUASH*, the hardware squashes and restarts one of the conflicting chunks (Figure 1(b)). In *STALL*, the hardware stalls the consumer chunk until the producer commits (Figure 1(c)). However, if the stall induces a cycle between two or more stalled chunks, the consumer is squashed. In *ORDER*, the hardware records the order of the two chunks, lets them continue and enforces the order at commit (Figure 1(d)). However, if the conflict forms a cycle between two or more ordered chunks, then one or more chunks are squashed. The three schemes are summarized in Table 2.

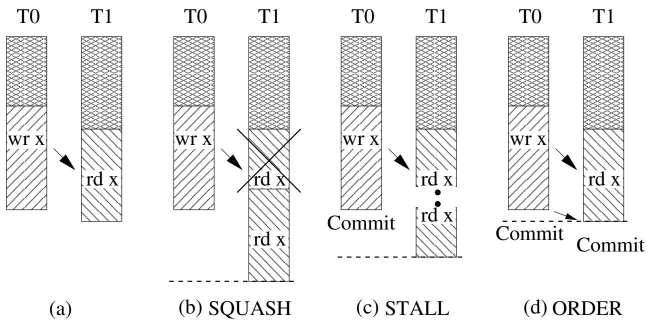

Figure 1. Conflict resolution schemes.

The main tradeoff is one of performance versus hardware cost. As we go from *SQUASH* to *STALL* and *ORDER*, we enable more

Design Point	Conflict Resol. Policy	Action on a Conflict Between Chunks
SQUASH	Eager Squash	Squash one of the conflicting chunks
STALL	Eager Stall	(1) Stall the consumer chunk; (2) if there is a cycle between two or more stalled chunks, squash the consumer chunk
ORDER	Eager Order	(1) Record the order of two chunks; (2) if there is a cycle between two or more ordered chunks, squash one or more chunks; (3) enforce the order at chunk commit

Table 2. Conflict resolution design points.

concurrency and, therefore, higher performance — as seen in Figure 1. However, the hardware is more costly and we need to keep more state. Specifically, *STALL* needs to record if a thread is stalled and, if so, which other thread stalled it, and detect cycles of stalled threads. *ORDER* needs to record if threads are currently ordered and, if so, by what type of dependence. In addition, it needs to enforce the commit ordering, and also detect cycles of dependent threads. Fortunately, thanks to the tight coupling of the contexts in an SMT, the hardware needed is simple and highly localized.

Next, we describe the three schemes. When we refer to a data dependence, we include RAW, WAR, and WAW, and they are at *memory line* granularity. The thread at the receiving end of the dependence is called consumer.

3.1.1. *SQUASH* Design

On a conflict, the chunk from one of the conflicting threads is squashed. Then, all of the processor resources used by the chunk (e.g., ROB entries, registers, and load/store queue entries) are released, and the cache lines written by the chunk are discarded (Section 3.2.1). Finally, the chunk restarts. We use the policy of oldest transaction wins, which helps make forward progress.

3.1.2. *STALL* Design

When a conflict is detected, the consumer chunk stalls before the actual consumer memory access is performed. The hardware records which chunk is stalled and which chunk is the producer one. When the producer chunk commits, the hardware resumes the consumer chunk, starting from the consumer memory access.

A chunk may stall on an already-stalled chunk. This is acceptable as long as the stalled chunks do not form a cycle. Consider Figure 2(a). In the figure, thread *T0* is about to write *x* and stalls on *T1*. Then, *T2* is about to write *y* and stalls on *T0*. This is fine because there is no cycle. Eventually, *T1* will commit and then *T0* will resume. When *T0* commits, *T2* resumes.

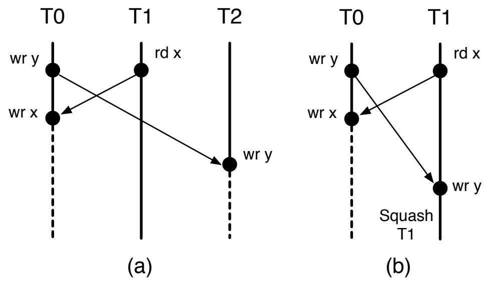

Figure 2. Examples of stalls.

When the hardware detects a cycle, it squashes the chunk that closes the cycle and resumes the chunks that were stalled on it. For example, in Figure 2(b), *T0* is stalled on *T1*. Then, *T1* attempts to write *y*, which would stall it on *T0*, creating a cycle. Consequently, *T1* gets squashed and *T0* resumes. A cycle can involve more than two chunks.

When BulkSMT stalls a chunk, the processor pipeline completes all the instructions in the chunk that are before the stalling one in program order. All the instructions that are after it must be flushed from the pipeline. Keeping them in the pipeline would lock up entries in resources that are shared by all of the contexts, such as the instruction queue. The result could be deadlock, as other contexts could fail to make progress. When the stalled chunk resumes, all of these instructions are reloaded again into the pipeline.

3.1.3. *ORDER* Design

When a conflict is detected, the hardware records the type and direction of the dependence. The chunks involved are allowed to proceed, but the hardware will enforce that they commit in the same order. The hardware also watches for a dependence that creates a cycle of ordered chunks (with two or more threads). If this happens, the hardware breaks the cycle by squashing and restarting one or more chunks — which may not include the one with the reference that closed the cycle.

To understand which chunks should get squashed in a cycle, consider the type of dependence. Figure 3 shows a RAW, WAW and WAR dependence and the squash rules that are easiest to support in hardware. Recall that a chunk squash also invalidates the cache lines updated by the chunk. In a RAW, one chunk wrote to the cache and a second one read. If we choose to squash the producer chunk, then we also have to squash the consumer. However, we can squash the consumer and not the producer. In a WAW, since dependences are at line granularity, if we choose to squash one of the chunks, we also have to squash the other. In a WAR, the squash of either one of the chunks does not cause the squash of the other. Section 3.2.5 uses these rules to decide which chunks to squash when a cycle is detected.

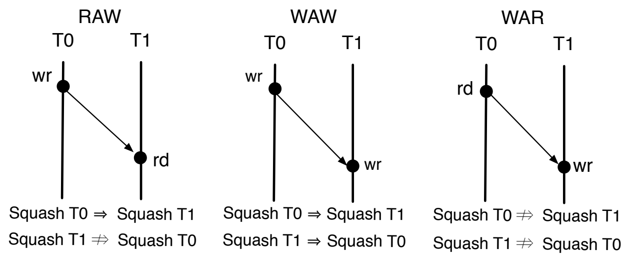

Figure 3. How squashes are affected by the type of dependence.

3.2. Basic Hardware Mechanisms

The basic mechanisms for BulkSMT operation are shown in Table 3. For each mechanism, the table shows its function, its implementation, and the designs it applies to. We consider each in turn.

3.2.1. Access Recording and Conflict Detection

Past work has used Bloom-filter based hardware signatures to detect conflicts between chunks or transactions executing on different cores (e.g., [5]). In BulkSMT, since the chunks are executing all in the same core and share the (multi-level) cache, it is easier to record the accesses with cache bits and use simple logic to

Mechanism	Function	Implementation	Designs
Access Recording and Conflict Detection	Record the addresses accessed by each chunk and detect when two chunks have a data conflict	*Access Bits* in cache and related logic	All (More in *ORDER*)
Cycle Detection	Record data conflicts and their ordering, and detect conflict cycles	*Dependence Table* and *Cycle Table*	*STALL* and *ORDER*
Advanced Conflict Recording	Represent the type of conflict between different chunks compactly	Enhanced Dependence Table	*ORDER*
Squash Set Generation	On a cycle of chunks with conflicts, decide the set of chunks to squash	Logic that operates on the Dependence Table	*ORDER*

Table 3. Basic mechanisms to support chunked execution in an SMT processor.

detect conflicts. Hence, we augment each cache line with a *Last Writer (LW)* context-ID, a read bit-mask with as many bits as contexts (*R[i]*), and a speculative bit (*Sp*). In an SMT with 4 contexts, this represents 7 bits per cache line (Figure 4(a)). When thread *k* reads from the cache, it sets *R[k]*; when it writes to the cache, it sets *Sp* and writes its context ID to *LW*. These bits are in the L1 and L2 caches and write buffers.

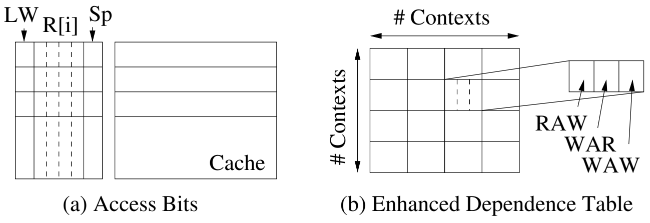

Figure 4. Two mechanisms for BulkSMT operation.

With this support, conflicts are detected as follows. When a load accesses a cache line, if *Sp* is set and *LW* is not the requester's ID, a RAW conflict is declared. When a store accesses a cache line, if the *R[i]* for any other context is set, a WAR is declared. Moreover, if *Sp* is set and *LW* is not the requester's ID, a WAW is (also) declared. In addition, when a chunk commits, the hardware performs two multi-cycle operations: (1) a flash clear of the *R[i]* bit corresponding to the chunk's context for all the cache lines, and (2) a flash conditional clear of the *Sp* bit of any cache line whose *LW* is equal to the committing context ID. Finally, when a chunk is squashed, the hardware performs two operations: (1) a flash clear of the *R[i]* bit corresponding to the chunk's context for all the cache lines, and (2) a flash conditional clear of the valid and *Sp* bits of any cache line whose *Sp* bit is set and the *LW* is equal to the squashed context ID. An example of SRAM cells augmented with similar support is shown in [3].

In a WAR dependence in *ORDER*, we may want to squash the writer chunk after it has written and not squash the reader chunk (Figure 3). In this case, we need to make sure that, after the squash of the writer (and invalidation of the lines it updated), we do not lose the record of any prior reader to those lines. To support this performance optimization, we add one additional bit per line called *Read_But_Missing* (RM). When a chunk is squashed, as the hardware invalidates a line, it checks the line's *R[i]*. If *R[i]* has a set bit for a thread *Ti* that is not being squashed, such bit is left unmodified and RM gets set. RM indicates that *R[i]* is up-to-date but the data is invalid. A future access to the line brings the line from memory, while clearing RM but not *R[i]* and, if applicable, recording a data conflict with *Ti*.

3.2.2. Cycle Detection

In both *STALL* and *ORDER*, we need a structure to record inter-thread data conflicts and their order — so that we stall the consumer thread in *STALL* and order the commit of producer and consumer in *ORDER*. Such structure also needs to detect conflict cycles. BulkSMT uses two low-cost hardware structures: the *Dependence Table (DT)* to record conflicts and the *Cycle Table (CT)* to detect conflict cycles. They are two-dimensional arrays, with as many rows and as many columns as hardware contexts in the processor. In the baseline design, each entry has one bit.

Figure 5(a) explains how they work. If there is a conflict where the chunk in thread T_i is the producer and the one in T_j is the consumer (represented as $T_i \rightarrow T_j$), the BulkSMT hardware sets the bits DT[i][j] and CT[i][j]. Every time that a new conflict is detected, the DT sets the corresponding bit. As the processor continues, in the background, the CT attempts to find if the new dependence has created a cycle. The CT does it by setting: (i) the bit corresponding to the latest conflict and (2) the bits corresponding to dependences *transitively* implied by all the recorded conflicts. A cycle is detected if a bit is set in the diagonal of the CT — i.e., a dependence has the same producer and consumer thread.

As an example, consider Figure 5(b). As conflict *d1* occurs, DT[0][1] and CT[0][1] get set. Later, as conflict *d2* occurs, DT[1][2] and CT[1][2] get set, and CT tries to find transitive dependences. This is done by taking the newest dependence (*d2*) and examining, in turn, its source and its destination, checking for other arrows connected there. Starting at the source (T_1), we consider all the arrows that point to it. In our example, the only one is *d1*. For this arrow, the transitive dependence is shown as *dA*. Specifically, any arrow whose destination is T_1 (i.e., *d1*) creates a new one (i.e., *dA*), whose source is unchanged and whose destination is the destination of the newest dependence (i.e., *d2*). In hardware terms, CT takes the column corresponding to T_1's ID (i.e., second column) and bit-ORs it into the column corresponding to the destination of the newest dependence *d2* (i.e., third column). This is shown in Figure 5(b).

The next step is to consider the arrows that start at the destination of the newest dependence and create transitive dependences. Our example does not have any. If it had (call it dependence *d3*), we would create an arrow from the source of *d2* to the destination of *d3*. Specifically, any arrow whose source is T_2 creates a new arrow whose destination is unchanged and whose source is the source of the newest dependence. In hardware, CT would take the row corresponding to T_2's ID (i.e., third row) and bit-OR it into the row corresponding to the source of dependence *d2* (i.e., second row).

The process described proceeds recursively: every time that a new transitive dependence is found in the CT, the algorithm proceeds to analyze its source and destination as described above to find new dependences. The process terminates when the CT no longer changes. Since an SMT processor has few contexts, CT is small (e.g., 4x4), and very few steps are typically needed.

Figure 5(c) shows an example of a cycle with two threads. On the left, we show the dependences *d1* and *d2* and, on the right, the evolution of the CT. When *d1* is flagged, bit CT[0][1] is set. When

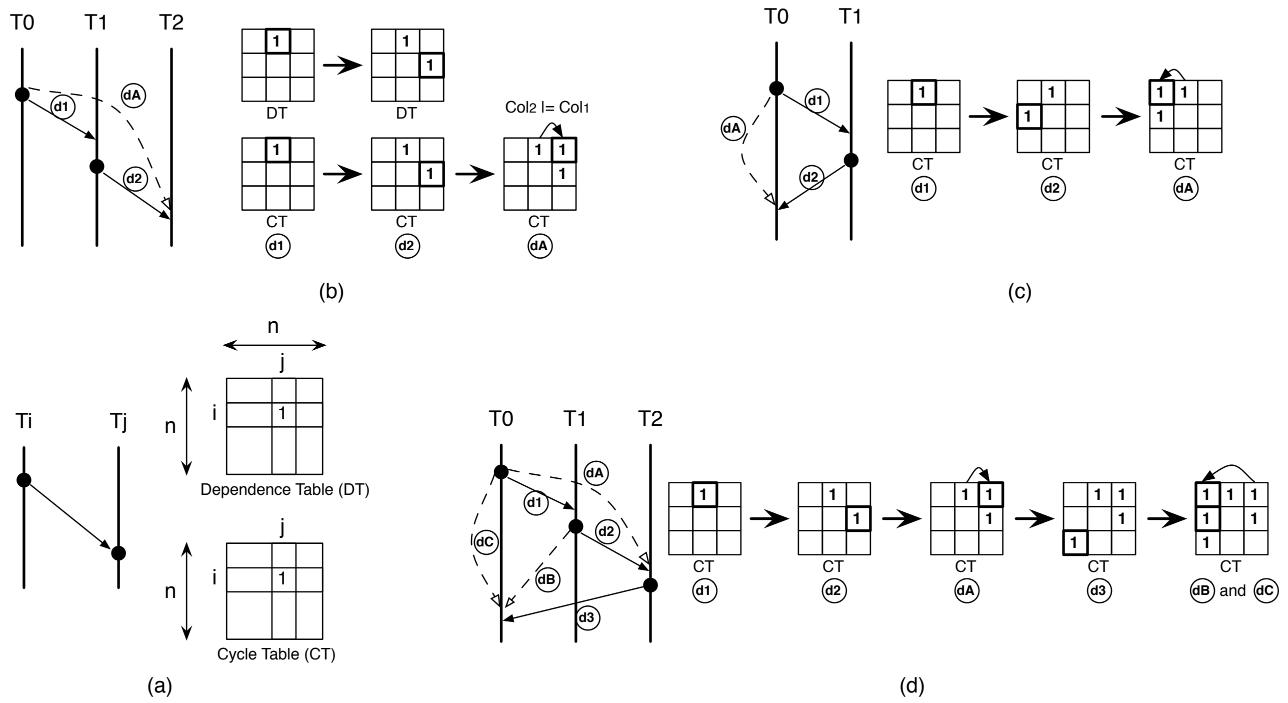

Figure 5. Operation of the Dependence Table and Cycle Table.

d2 is flagged, bit CT[1][0] is set and the algorithm proceeds by ORing the second column into the first one. The set bit CT[0][0] flags the cycle, which corresponds to arrow *dA*. While the algorithm stops as soon as it finds a cycle, for completion, we note that there is another cycle. It is an arrow not shown in the figure that goes from T_1 to T_1. It is obtained by processing the destination of *d2*. It appears as we bit-OR the first row into the second row and bit CT[1][1] gets set.

Figure 5(d) shows a three-thread cycle. Dependence *d1* sets CT[0][1]; dependence *d2* sets CT[1][2] and uncovers *dA*, setting CT[0][2]; finally, dependence *d3* sets CT[2][0] and uncovers *dB* and *dC*, setting CT[1][0] and CT[0][0] — hence flagging a cycle.

3.2.3. Additional Issues Related to the DT and CT

The CT is not a time-critical structure. While the DT must be updated as soon as the dependence occurs, the CT can buffer its inputs and only later get updated and run the cycle detection algorithm. It is always correct to find a few cycles later that a cycle occurred. At that point, the Squash Set Generation algorithm (Section 3.2.5) will be run based on the *up-to-date state* of the DT.

The DT and CT are also updated when a chunk commits or gets squashed. Consider first that the chunk in thread T_i is ready to commit. In *ORDER*, BulkSMT first checks if it can commit. If any bit in column *i* of the DT is set, the thread has to stall — and post that it is stalled. In all other cases of *ORDER* and *STALL*, the chunk commits and then BulkSMT runs the algorithm of Figure 6. Similarly, after the chunk in thread T_i is squashed, BulkSMT runs the algorithm of Figure 6.

3.2.4. Advanced Conflict Recording

In *STALL*, each DT entry only needs to record if there is a dependence or not. Therefore, one bit per entry suffices. In *ORDER*, each DT entry also needs to record what type(s) of dependence there are between the two chunks — RAW, WAW, or WAR. This information

```
1) Wake up any chunks that are stalled and can now proceed:
   1.1) Consider row i in the DT. Find all columns where their only
        set bit is in row i
   1.2) For each column j in this set
        1.2.1) If in STALL: wake up thread $T_j$
        1.2.2) If in ORDER: if thread $T_j$ is stalled, wake up thread $T_j$
2) Clear row DT[i][.] and column DT[.][i]. Clear the CT
3) Copy the DT to the CT. Regenerate all the transitive dependences in CT
```

Figure 6. Actions at the commit/squash of thread T_i's chunk.

is needed in case of a cycle, to decide what chunks to squash (Section 3.2.5). Recall that the dependence type impacts which chunk to squash (Figure 3).

Two chunks may have multiple dependence types (on the same or different variables). Hence, in *ORDER*, the DT has three bits per entry, one per each type of dependence (Figure 4(b)). Note that the CT is unaffected, and it still has one bit per entry. The bit in the CT entry is set if any of the three bits in the DT entry is set.

3.2.5. Squash Set Generation

In *ORDER*, when the CT detects a cycle, the hardware stalls the processor and uses the DT (which is consistent with the current speculative memory state) to decide which chunks to squash to break the cycle. The algorithm used to select such chunks is called *Squash Set Generator (SSG)*. It reads the bits currently in the DT and applies the rules of Figure 3 for RAW, WAW, and WAR dependences.

The Baseline SSG algorithm starts by putting in the set of chunks to squash (the squash set) the chunk that closed the cycle. Then, it follows forward dependences from that chunk, using the rules in Figure 3 to put additional chunks in the squash set. The squash propagation stops when forward dependences either bring us to chunks already in the set or they do not propagate squashes because they are WAR dependences. Then, SSG goes back to the orig-

Event	Actions
Chunk wants to commit	Initiate a global commit; when it succeeds, commit locally **EE:** Wait for the completion of all the buffered previous memory accesses, then perform local commit **LL:** Send signature out, wait for the global confirmation, then perform local commit. For performance, perform commit combining under *ORDER*
Reception of a coherence event that may cause a squash	Squash any local chunk that needs to be squashed (even the stalled ones) **EE:** Use the address of the coherence message to index the cache and read the Access Bits; if a conflict [14] is detected, squash the corresponding local chunk(s) **LL:** Intersect the incoming signature with local signatures [6]; if intersection is not null, squash the corresponding local chunk(s) Propagate the squash inside the SMT processor

Table 4. Rules for integrating local and global chunk-based protocols.

inal chunk and follows backward dependences from there, again using the rules. The backward propagation stops when we reach chunks already in the set or the dependences do not propagate the squash because they are RAW or WAR dependences. The algorithm is recursive.

Figure 7 shows an example of a cycle with three chunks. The read in thread *T0*'s chunk closes a cycle. Consequently, *T0* is put in the squash set. From *T0*, SSG then follows the RAW to thread *T1*, which is also put in the set. The next forward dependence is a WAR to *T2*, which stops the propagation. Then, SSG goes to *T0* and propagates backward. Since we find a RAW to *T2*, back-propagation stops. Consequently, only *T0* and *T1* get squashed.

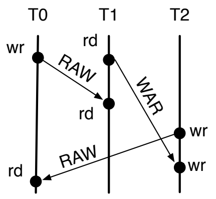

Figure 7. Example of breaking the cycle of ordered chunks.

In reality, a given cycle can be broken in multiple ways, possibly resulting in different numbers of squashed chunks. Consequently, the Advanced SSG algorithm does not simply squash the chunks found in the squash set as described after the first try. Instead, it then picks each chunk in the set in turn and re-runs the algorithm starting from that chunk. These new runs may result in fewer chunks to squash. For example, in Figure 7, if SSG starts from *T1*, it finds that it only needs to squash *T1* to break the cycle. Consequently, the Advanced SSG algorithm breaks the cycle in the way that minimizes the number of squashed chunks.

Since over 95% of the cycles that we found only involve two chunks, the Advanced SSG adds very little overhead. Moreover, Section 5 shows that the Advanced SSG helps *ORDER* handle high-contention locks.

After BulkSMT has squashed the chunks to break the cycle, it uses the resulting DT to regenerate the CT. If the CT finds that there is still a cycle, the whole process is repeated. The CT may still find a cycle if the last dependence recorded ended up creating two cycles, and with the use of the Advanced SSG algorithm, we broke only one. Overall, we use the Advanced SSG algorithm. Its hardware cost is modest, since it only accesses the DT and the number of contexts in an SMT is fairly small. Moreover, it only runs in the relatively rare case of a cycle.

4. Chunked-Execution Multicores of SMTs

SMT cores with chunked-execution support should be amenable to integration into multicores and multi-socket systems. We now examine the additional microarchitecture needed to use BulkSMT as a building block for a chunked-execution multicore. In our discussion, we refer to the hardware actions across SMT cores as *global*, while those across the contexts of an SMT core as *local*.

There has been much research on designing multiprocessor hardware that supports chunks or transactions using single-context (non-SMT) cores (e.g., [6, 7, 9, 14, 16, 17]). To cover a broad design space, we consider two global designs. The first one (*EE*) uses eager version management and eager conflict resolution, and is like LogTM [14]. The other (*LL*) uses lazy version management and lazy conflict resolution, and is like BulkSC [6]. Next, we outline the relevant parts of the two global protocols and then describe the integration with the local protocol.

4.1. Global Protocols Examined

The EE scheme uses the Access Bits in the caches to flag conflicts between threads running on different cores. As in LogTM, we need to augment each core with hardware-based undo logs that save the old values when a speculative thread writes a variable. Since we use SMT cores, a core has as many undo logs as hardware contexts.

The LL scheme could also use the Access Bits in the caches to flag inter-core conflicts like TCC [9]. However, since we model BulkSC [6], we use hardware-based address signatures to detect conflicts. Hence, in an SMT core, each context has a R an a W signature. When chunks commit, they send out their signatures, which are intersected with those in the receiving cores. To detect conflicts between the contexts of a core, we still use the Access Bits in the caches.

4.2. Integrating the Local and Global Protocols

Table 4 lists our two rules for a design that integrates local and global chunk-based protocols. The first rule applies when a chunk wants to commit: it should first initiate a commit globally (across cores) and, when it succeeds, commit locally among the threads in the SMT. In the EE scheme, this implies waiting for the completion of all the buffered previous memory accesses by the chunk, and then performing the local commit in the SMT core. In the LL scheme, it implies sending the chunk's signature out to the global network, waiting for the global commit confirmation, and then performing the local commit in the SMT core.

In the LL scheme, since commit is costly, we augment *ORDER* with *Commit Combining*. This event occurs when a consumer chunk completes execution before its producer chunk in the same SMT core does. The consumer has to wait to commit until after the producer commits. With commit combining, when the producer completes, both chunks perform the commit together — i.e., they send a combined signature out and then commit locally together.

The second rule applies when a core receives a coherence event that may cause a squash. The core must check against all of the local chunks — even the stalled ones. In the EE scheme, this means using the address of the coherence message (e.g., invalidation) to index the cache and check the Access Bits; if a conflict is found, we squash the corresponding chunk(s). In the LL scheme, the hardware intersects the incoming signature with all the local signatures; if an intersection is not null, we squash the corresponding chunk(s). Moreover, in both EE and LL, the squash needs to be *propagated*. Specifically, in *ORDER*, we run the SSG algorithm (Section 3.2.5) to detect other chunks to squash; in *STALL*, we wake up all the chunks that are stalled waiting *only* on the squashed chunks.

5. Implementation Issues

5.1. Cycle Detection Algorithm Implementation

To complement the description of the cycle detection algorithm, we outline its hardware implementation in a Cycle Table Module (Figure 8). The module contains the Cycle Table (CT), combinational logic to perform the steps of the cycle detection algorithm (*Dependence Generator*) and detect a cycle (*Cycle Checker*), and the *Shadow Cycle Table* (SCT). The latter is a table like the CT that contains temporary state as the algorithm runs.

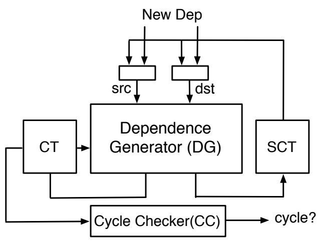

Figure 8. Cycle Table Module.

In idle state, CT holds a certain bit pattern and SCT is clear. When the program generates a new dependence, it is encoded as a (*src*,*dst*) code and goes through the Dependence Generator. The latter finds if the dependence itself sets a new bit in CT. If so, the new bit is set in both CT and SCT, and the multi-step process of finding transitive dependences starts. Such process involves the SCT feeding the new bit to the Dependence Generator in two steps: (1) first to combine with existing dependences at the *source* of the new dependence (which triggers the Dependence Generator to bit-OR two CT columns as in Figure 5(b)) and (2) then to combine with dependences at the *destination* of the new dependence (which triggers the Dependence Generator to bit-OR two CT rows). After these two steps, the bit is cleared from the SCT.

If, in any of these two steps, a bit that was not set in CT gets set, we have found a new dependence by transitivity. Such bit is set in both CT and SCT, and the SCT will process the new bit once it is done with the first one. The process continues until no new dependence is found and SCT becomes clear. At all times, the Cycle Checker uses simple logic to check if a bit gets set in CT's diagonal. If one gets set, a cycle is flagged and the whole process stops.

If, during this process, the program generates a new dependence, the dependence is buffered. Completing the current process of uncovering all transitive dependences has higher priority. The new dependence will be processed immediately after.

Based on this description, Figure 9 lists the hardware cost of the Cycle Table (CT) Module, and of all the other hardware structures required by BulkSMT. We assume a BulkSMT processor with n contexts. The CT Module only needs two arrays of n^2 bits and the combinational logic for the Dependence Generator (DG) and Cycle Checker (CC). The Enhanced Dependence Table (DT) needs an array of $3 \times n^2$ bits. Finally, the Access Bits need $n + log_2 n + 1$ bits per cache line. Overall, the hardware requirements of BulkSMT are very modest.

Number of Contexts in the BulkSMT Processor: n			
Structure Name		Cost	Hardware Type
CT Module	CT	n^2	Array of bits
	SCT	n^2	Array of bits
	CC + DG	Combinat. logic	—
Enhanced DT		$3 \times n^2$	Array of bits
Access Bits per cache line		$n + log_2 n + 1$	Extension to tags

Figure 9. Hardware requirements of the BulkSMT mechanisms.

5.2. Cache Conflicts

While our discussion has focused on chunk squashes due to dependences, chunks may also get squashed on cache overflow. Specifically, when a cache conflict displaces a line with non-null Access Bits from the lowest cache level, the chunks that accessed the line get squashed. In addition, in *ORDER*, the hardware follows the rules of Figure 3 to find dependent chunks to squash. Finally, after the squashes, in both *STALL* and *ORDER*, chunks that were stalled on the squashed ones are released. Given the potential cost of these actions, it may be beneficial to tune the cache replacement algorithms to avoid these cases. In this paper, we have not done so.

5.3. Handling High-Contention Synchronizations

High-contention synchronizations are a concern for chunked-execution architectures because they introduce frequent dependences between chunks. Such dependences cause squashes or stalls. One way to minimize their impact is to explicitly terminate the chunk with a software command after or before a high-contention synchronization. The shortcoming of this approach is that it needs either a profiling pass to identify high-contention synchronizations or support to learn the frequent dependences dynamically.

In this paper, we do not use a profiling pass or hardware to learn the frequent dependences dynamically. Hence, we do not terminate chunks in software at high-contention synchronizations. The one exception is at barriers: since it is clear that chunks conflict at barriers, we place chunk termination commands inside the barrier library call or macro. By terminating the chunk, we ensure the work before the barrier is not squashed due to a conflict in the barrier.

For an interesting illustration of how chunked-execution works, consider a high-contention lock under *ORDER*. BulkSMT uses Test&Test&Set for the lock. In Figure 10(a), the chunk in thread *T0* grabs the lock. Then, *T1* spins on it, creating a RAW dependence. When *T0* releases it, it creates a cycle. The Advanced SSG algorithm (Section 3.2.5) squashes the minimum number of chunks to break the cycle, namely just the one in *T1*. As the chunk restarts (Figure 10(b)), an ordered dependence is created, which still allows both chunks to eventually commit successfully.

6. Evaluation Setup

In our evaluation, we model a 4-context BulkSMT core alone or in a 4-core multicore chip. We model all combinations of *SQUASH*,

Configurations Used: XX-YY, where:
XX: Type of core: SQ: 4-context *SQUASH* BulkSMT design ST: 4-context *STALL* BulkSMT design OR: 4-context *ORDER* BulkSMT design BK: Single-context core with chunked-execution support
YY: Type of global (inter-core) protocol: EE: Eager version management and eager conflict resolution LL: Lazy version management and lazy conflict resolution

Figure 11. Names of configurations used.

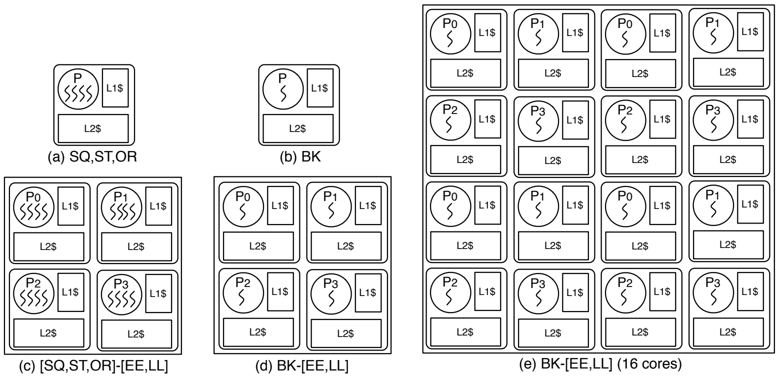

Figure 12. Configurations used.

Core	Memory Subsystem	Chunk Parameters
Frequency: 5.0 GHz # of contexts: 4 for BulkSMT, 1 for *BK* Fetch/issue/comm width:4/4/5 I-window: 80 ROB: 176 (1/4 per thread in BulkSMT) LdSt/Int/FP units: 2/3/3 Ld/St queue: 56 (1/4 per thread in BulkSMT) Int/FP registers: 96/80 Branch penalty: 17 cyc (min)	Private write-back D-L1: Size/assoc/line: 32KB/4-way/32B Hit round trip: 2 cycles Private write-back L2: Size/assoc/line: 256KB/8-way/32B Hit round trip: 9 cycles L2 miss delay: Hit other L2s (avg): 16 cyc in 4 cores/chip 20 cyc in 16 cores/chip To memory: 500 cycles round trip	# of outstanding chunks/thread: 1 Target chunk size: 10k instructions Commit latency: 50 cycles (1 core) 200 cycles (4 cores) 250 cycles (16 cores) Signature size: 2K bits R and W Signature config: S14 from [5]

Table 5. Simulated system configurations.

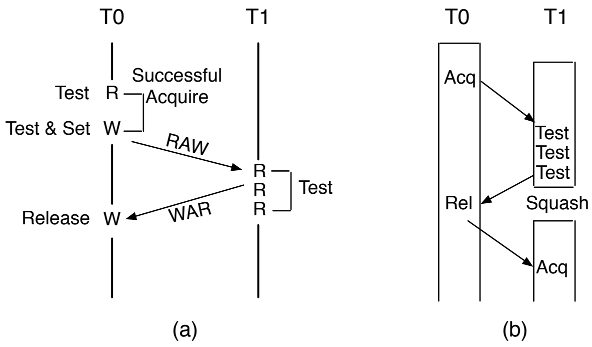

Figure 10. Competing for a high-contention lock in *ORDER*.

STALL, and *ORDER* with the *EE* and *LL* global protocols. In addition, we compare the performance to machines built out of single-context cores with chunked-execution support called *BK*. The configurations used are shown in Figure 11 and Figure 12.

We evaluate these designs using a cycle-accurate execution-driven simulator based on SESC [19] with detailed models for the processor, the memory subsystem and the interconnect. The architectural parameters are shown in Table 5. The BulkSMT and *BK* cores have the same issue width and hardware structure sizes. However, in BulkSMT, the ROB and load-store queue are partitioned equally among the 4 contexts. Each core has private L1 and L2 caches. The global protocol is similar to BulkSC's [6] in *LL* and LogTM's [14] in *EE*.

We use the applications from SPLASH-2 and PARSEC that have a noticeable degree of interactions between threads. From SPLASH-2, the applications and inputs we use are: Barnes (16k particles), Cholesky (tk29.0), Ocean (258x258 ocean), Radiosity (room), Radix (256K keys) and Raytrace (car). From PARSEC, they are: fluidanimate (simmedium) and streamcluster (simmedium). For the other applications, the total squash time is very small and the different core designs discussed make no difference. The applications run with 1, 4 or 16 threads. The applications are dynamically broken down into chunks of 10K dynamic instructions automatically in hardware. However, the software places chunk termination commands inside the library calls for barriers, to minimize any work squashed at barriers. We use these relatively large chunks because, as discussed in [1], they more accurately represent future uses of chunked architectures, where the compiler optimizes the code, and the commit cost is more effectively amortized.

7. Evaluation

7.1. Performance Comparison

We want to find out which of the BulkSMT designs performs best, and how does the performance of BulkSMT and *BK* compare (i) for a fixed number of cores (which is a proxy for the amount of hardware) and (ii) for a fixed number of total threads. Due to space limitations, we do not compare a chunked-execution platform to a non-chunked one. Our focus is chunked-execution environments and our goal is to find out the impact of SMT on them.

In our plots, we break an application's execution time into the following types of processor cycles: cycles retiring instructions (*Useful*), stalled due to pipeline hazards (*ProcPipe*), stalled due to memory accesses (*ProcMem*), stalled because the chunk is stopped in *STALL* or *ORDER* (*ChunkStall*), and performing work that will be squashed (*Squashed*). The total time that a processor is stalled while committing a chunk is negligible. The plots also show the geometric mean of the applications, which cannot be broken down.

7.1.1. Comparing BulkSMT Designs

Figure 13 compares the execution time of all of the 4-core architectures: BulkSMT designs running with 16 threads as in Figure 12(c) and *BK* designs running with 4 threads as in Figure 12(d).

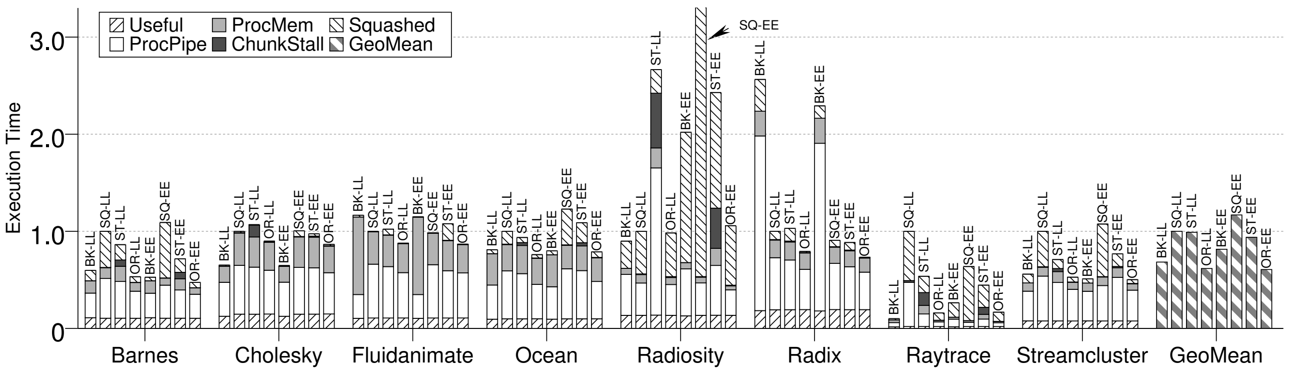

Figure 13. Execution time of 4-core architectures. The BulkSMT designs (SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, OR-EE) run with 16 threads, while the *BK* designs (BK-LL, BK-EE) run with 4 threads. In each application, the bars are normalized to SQ-LL. The SQ-EE bar for Radiosity reaches 4.23.

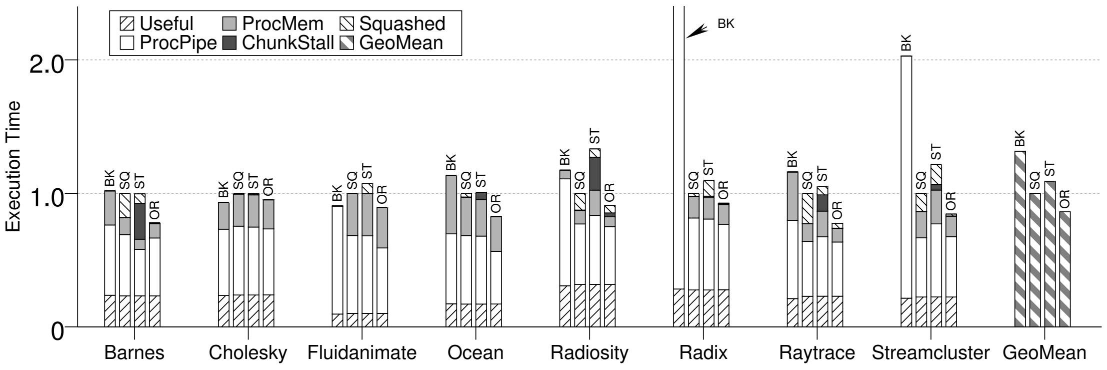

Figure 14. Execution time of single-core architectures. The BulkSMT designs (SQ, ST, OR) run with 4 threads, while the BK design runs with 1 thread. In each application, the bars are normalized to SQ. The BK bar for Radix reaches 3.39.

Each application has 8 bars, organized as LL first and EE later, all normalized to SQ-LL.

Comparing the different BulkSMT designs, we see that *SQUASH* suffers from *Squashed* time, since it is not tolerant of dependences. As we move to *STALL*, *Squashed* decreases, but some *ChunkStall* time appears — often resulting in faster execution. Finally, as we move to *ORDER*, both *Squashed* and *ChunkStall* largely disappear, resulting in the fastest design. These trends are clearest in Barnes and Raytrace.

In Radiosity, the large changes across bars are due to the frequent enqueue and dequeue operations in a task queue. Each operation involves the update of shared variables in critical sections, which translates into squash and stall in *SQUASH* and *STALL*.

The particular characteristics of each application determine whether the LL or EE designs are better.

Overall, *ORDER* is the recommended design. On average, its LL design reduces the execution time by 38% relative to *SQUASH* or *STALL*. In the EE environment, the reductions attained by *ORDER* are 48% relative to *SQUASH* and 35% relative to *STALL*.

7.1.2. BulkSMT vs BK for a Fixed Number of Cores

We return to Figure 13 to compare the BulkSMT and *BK* designs. They use the same core count, which is a proxy for hardware amount, although BulkSMT runs with 16 threads and *BK* with 4.

Since the applications run with more threads in BulkSMT, they can attain higher performance. Moreover, the tightly-coupled SMT hardware enables fast inter-thread communication. However, these applications do not exhibit linear speedup curves up to 16 threads. Instead, their speedups saturate. Moreover, more inter-thread dependences appear, which the BulkSMT designs have to handle. Finally, with BulkSMT, multiple threads compete for the fixed resources of a core. The relative impact of these factors determines the execution time.

For example, in Radix, the BulkSMT designs perform better. With a single context executing in each core, processor resources are underutilized because the ILP is low; when 4 contexts are executing per core, processor resources are utilized better. On the other hand, in Raytrace, BulkSMT designs perform worse because of the increased contention for locks among the more threads.

Looking at the geometric mean, we see that *ORDER* is faster than *BK*, although *SQUASH* and *STALL* are not. Specifically, in the EE environment, *ORDER* reduces the execution time of the applications by an average of 26% compared to *BK*. The corresponding number in the LL environment is 10%.

Figure 14 repeats the experiments for 1 core, such that the BulkSMT designs run with 4 threads as in Figure 12(a), and *BK* with 1 as in Figure 12(b). Each application only has 4 bars because there are no EE or LL effects. The bars are normalized to SQ.

We largely observe the same trends as in Figure 13, except that the BulkSMT designs perform relatively better than *BK*. The reason is that the applications scale much better from 1 to 4 threads than from 4 to 16. From the mean, *ORDER* is the best design, followed

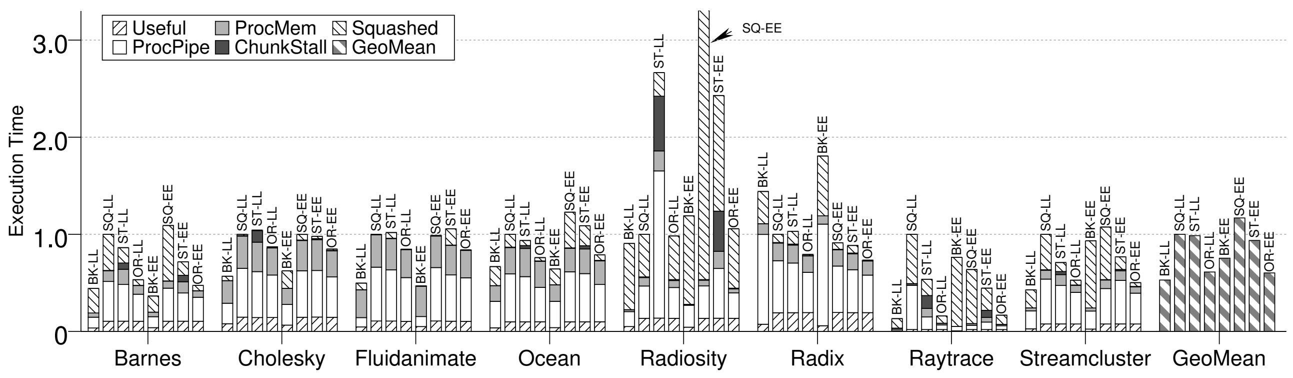

Figure 15. Execution time of different 16-thread architectures. The BulkSMT designs (SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, OR-EE) use 4 cores, while the *BK* designs (BK-LL, BK-EE) use 16 cores — hence about 4 times more hardware. In each application, the bars are normalized to SQ-LL. The SQ-EE bar for Radiosity reaches 4.23.

by *SQUASH*, *STALL*, and *BK*. On average, *SQUASH*, *STALL*, and *ORDER* reduce the execution time of the applications by 23%, 17%, and 32%, respectively, relative to *BK*. Hence, supporting BulkSMT is cost effective.

7.1.3. BulkSMT vs BK for a Fixed Number of Threads

Figure 15 shows the execution time of 16-threaded architectures, where BulkSMT designs use 4 cores as in Figure 12(c) and *BK* designs use 16 cores as in Figure 12(e). As a result, the *BK* designs use about 4 times more hardware. The figure is organized as usual.

The figure shows that the OR designs, with much less hardware than the *BK* systems attain, on average, about the same performance as the *BK* systems. Specifically, the average execution time of OR-LL is 15% higher than that of BK-LL, while OR-EE's execution time is 20% lower than BK-EE's. The other BulkSMT designs are slower.

Comparing the *BK* designs of Figures 13 and 15, we see that, as applications move from 4 to 16 threads, they reduce their *Useful* and other stall times. However, they often increase their *Squashed* time. On the other hand, *ORDER* often avoids squashes thanks to its technique of ordering chunks. This is the case for Radiosity, Radix, Raytrace and Streamcluster.

Overall, combining all the findings in this performance section, we conclude that the *ORDER* BulkSMT design is attractive. For 16-threaded applications, it performs significantly better than single-context core platforms with the same core count, and performs about the same as single-context core platforms with four times more hardware.

7.2. Dependence Analysis in *STALL* and *ORDER*

Figure 16 shows the number of dependences (*WAW*, *RAW* and *WAR*) observed between the 4 threads running on a BulkSMT core. The data corresponds to the 16-thread ST-LL and OR-LL environments. We do not show data for *SQUASH* because, on a dependence, one of the threads gets squashed. We also do not show data for the EE environment because the trends are qualitatively similar. For each application and architecture, the bars are normalized to 1 and broken down into the type of dependence. The number on top of each bar is the average number of dependences per 100K instructions.

We see that the number of dependences in *ORDER* is larger than in *STALL*. This is because, in *STALL*, one of the chunks is

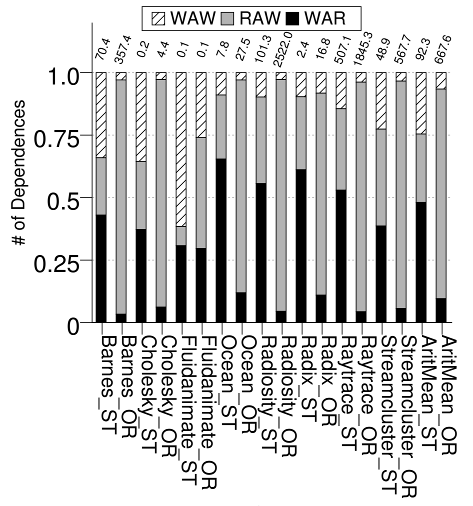

Figure 16. Types of dependences.

stopped after the dependence. In *ORDER*, both chunks can continue execution. Therefore, more dependences can be established between the two chunks. We also see that applications with a large difference between the number of dependences in *STALL* and *ORDER* have a much faster *ORDER* architecture than *STALL* in Figure 13. This is because many same-direction dependences between two concurrently-executing chunks are formed in *ORDER*, while *STALL* has to stall. Finally, we see that the dominant dependence type in *ORDER* is RAW, while the three types of dependences are more equally distributed in *STALL*.

7.3. Dependence Cycles in *ORDER*

Figure 17 characterizes the dependence cycles observed between the 4 threads running on a BulkSMT core. The data corresponds to the 16-thread OR-LL environment. For each application, the bars are normalized to 1 and broken down into the different types of cycles. The large majority of the cycles are formed between two chunks, and are classified according to the type of dependence. For example, RAW $\rightarrow$ WAR means that the cycle is formed by a RAW dependence followed by a WAR one in the opposite direction. Accordingly, there are 9 types of cycles between two chunks. The topmost class is cycles with more than two chunks. On top of each bar, we show the number of cycles per 100K instructions.

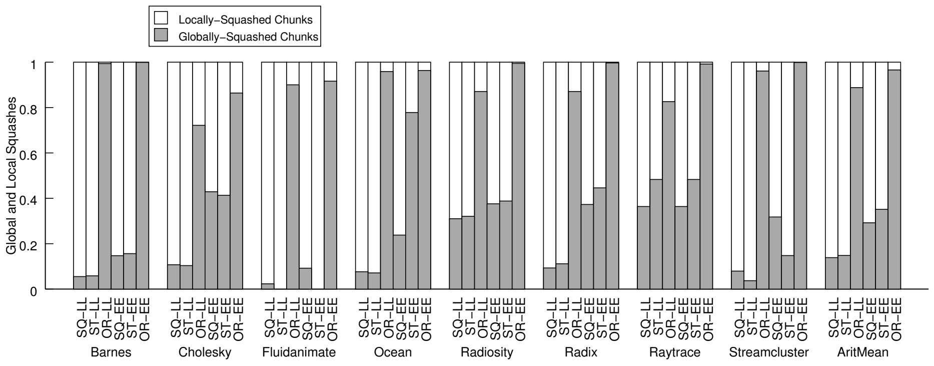

Figure 18. Comparing the number of locally-squashed and globally-squashed chunks.

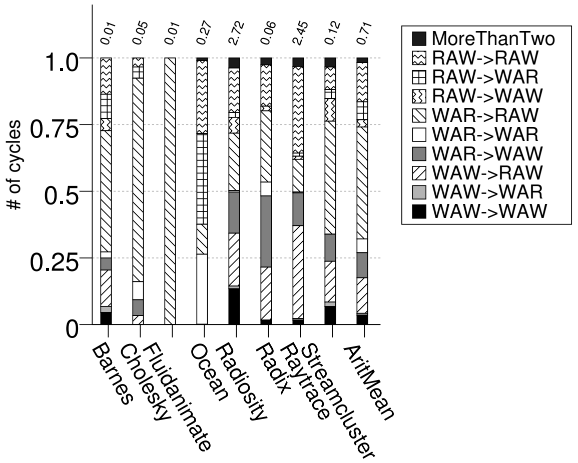

Figure 17. Types of cycles.

Some of the major types of cycles are WAR → RAW (where two consecutive reads are interleaved by a remote write) and RAW → RAW (where information is transferred from one processor to another and then back to the first one). Comparing Figure 17 and Figure 16, we see that the average number of cycles is typically much lower than the average number of dependences.

7.4. Squash Set Size

Figure 19 shows the number of chunks that need to be squashed to break a cycle (i.e., the Squash Set size) in *ORDER*. The figure corresponds to 16-thread OR-LL. For each application, the bar is normalized to 1 and broken down into squash set sizes: one (*SqSet:1*), two (*SqSet:2*), three (*SqSet:3*), or four (*SqSet:4*). We see that, in practially all cases, only one chunk is squashed to break the cycle; the other chunk(s) can continue.

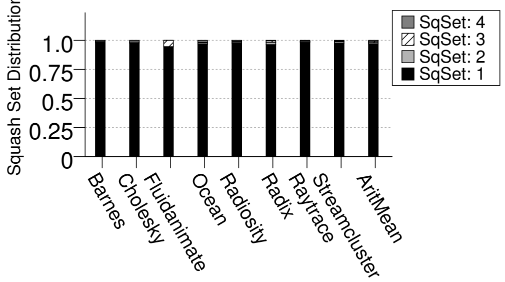

Figure 19. Squash sets sizes.

7.5. Comparing Local vs Global Squashes

Figure 18 compares the number of chunks squashed by intra-SMT conflicts (*Locally Squashed*) to those squashed by inter-core conflicts (*Globally Squashed*). The figure shows data for each of the BulkSMT designs with 16 threads: SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, and OR-EE. There is a bar for each design, normalized to 1, and broken down into locally- and globally-squashed chunks.

The figure shows that, while most of the squashes in *SQUASH* and *STALL* are local, the opposite is true for *ORDER*. *ORDER*'s ability to allow the two chunks involved in a dependence to continue executing enables it to eliminate practically all of the local squashes. Interestingly, *STALL* still suffers local squashes. The reason is that stalled chunks, as they wait, are effectively vulnerable to squashes due to new dependences that appear.

8. Related Work

Beyond the chunked-execution architectures listed in Section 2, the most relevant work includes techniques that try to increase the concurrency of conflicting transactions in hardware transactional memory systems. There are two proposals, DATM [18] and SONTM [2], which apply to multicore systems with single-context processors.

DATM manages the dependences between uncommitted transactions, sometimes forwarding data between them to be able to safely commit conflicting transactions. It uses a bus-based shared-memory machine and proposes the FRMSI snoopy-based cache coherence protocol. This is a new protocol with 11 stable states. Such protocol supports the forwarding of lines between caches like an update-based protocol. It also needs to select the correct version of a datum among the several that exist in the different caches of the machine. It has per-word access bits to support the ability to merge cache lines that have been partially updated by different processors. Finally, to keep the order of transactions, it has an order vector of transactions stored in each cache.

The SONTM system maintains an upper bound and a lower bound Serializability Order Number (SON) for each transaction. They are updated when a transaction performs a memory operation and when a transaction commits. During a transaction's execution, when the upper bound is smaller than the lower bound, the transaction is aborted because it cannot be serialized with other dependent transactions. While SONTM does not modify the cache coherence protocol, it adds substantial overhead. Specifically, each memory location accessed has a read-number and a write-number stored in memory. While some optimizations are possible, each load and store instruction needs to get the read-number or write-number to potentially update the upper and lower bounds. Moreover, a validation step at a transaction commit involves broadcasting write-numbers of all the updated data and receiving read-number responses from other processors.

Overall, compared to these schemes, we focus on optimizing dependent chunks executing on the same core, rather than across cores. Hence, our hardware is substantially simpler and has much less overhead.

Tullsen et al. [22] present a hardware structure called Lock Box that allows threads in an SMT to synchronize efficiently. A thread trying to acquire a lock is blocked if the lock is busy. When a thread releases the lock it wakes up the blocked thread. *STALL* proposes a similar idea for any data dependence.

9. Conclusions

None of the previously-proposed architectures that continuously execute chunks of instructions or transactions use SMT cores — although SMT cores are widely deployed and would likely be used in a commercial implementation of these architectures.

To address this problem, this paper has presented the first SMT design that supports continuous chunked execution. The design, called *BulkSMT*, can be used either in a single-core processor or in a multicore of SMTs. We have proposed three BulkSMT configurations with different cost and performance: *SQUASH*, *STALL*, and *ORDER*. We have described a set of novel architectural primitives that enable chunked execution in an SMT core. Finally, we have shown how to augment the resulting SMT core to work in a multicore of SMTs that supports chunked execution. Our results, based on simulations of SPLASH-2 and PARSEC codes, showed that BulkSMT supported this mode of execution cost-effectively. For example, in a 4-core multicore with eager chunked execution, BulkSMT reduces the execution time of the applications by an average of 26% compared to running on single-context cores. The corresponding number for lazy chunked execution is 10%. In a single-core machine, the average execution time reduction is 32%.

References

[1] R. Agarwal and J. Torrellas. FlexBulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes. In *International Symposium on Computer Architecture*, June 2011.

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

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

[4] C. Blundell, A. Raghavan, and M. M. K. Martin. RetCon: Transactional Repair Without Replay. In *International Symposium on Computer Architecture*, June 2010.

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

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

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

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

[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 *International Symposium on Computer Architecture*, June 2004.

[10] R. Kalla. POWER7: IBM's Next Generation POWER Microprocessor. In *HotChips 21*, August 2009.

[11] D. Koufaty and D. Marr. HyperThreading Technology in the NetBurst Microarchitecture. In *IEEE Micro*, March/April 2003.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *International Symposium on Computer Architecture*, June 2008.

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

[14] K. Moore, J. Bobba, M. J. Moravam, M. Hill, and D. Wood. LogTM: Log-based Transactional Memory. In *International Symposium on High Performance Computer Architecture*, February 2006.

[15] S. Pant and G. Byrd. A Case for Using Value Prediction to Improve the Performance of Transactional Memory. In *TRANSACT*, June 2009.

[16] S. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *PACT*, 2008.

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

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

[19] 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.

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

[21] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture*, June 1995.

[22] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *International Symposium on High Performance Computer Architecture*, February 1999.

[23] 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.

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