Appeared in IEEE/ACM International Symposium on Computer Architecture, June 2006 (ISCA'06)

Bulk Disambiguation of Speculative Threads in Multiprocessors*

Luis Ceze, **James Tuck**, **Călin Caşcaval**† and **Josep Torrellas**

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

†IBM T.J. Watson Research Center
cascaval@us.ibm.com

Abstract

Transactional Memory (TM), Thread-Level Speculation (TLS), and Checkpointed multiprocessors are three popular architectural techniques based on the execution of multiple, cooperating speculative threads. In these environments, correctly maintaining data dependences across threads requires mechanisms for disambiguating addresses across threads, invalidating stale cache state, and making committed state visible. These mechanisms are both conceptually involved and hard to implement.

In this paper, we present *Bulk*, a novel approach to simplify these mechanisms. The idea is to hash-encode a thread's access information in a concise signature, and then support in hardware signature operations that efficiently process sets of addresses. Such operations implement the mechanisms described. Bulk operations are inexact but correct, and provide substantial conceptual and implementation simplicity. We evaluate Bulk in the context of TLS using SPECint2000 codes and TM using multithreaded Java workloads. Despite its simplicity, Bulk has competitive performance with more complex schemes. We also find that signature configuration is a key design parameter.

1. Introduction

In recent years, efforts to substantially improve the programmability and performance of programs have resulted in techniques based on the execution of multiple, cooperating speculative threads. Such techniques include Transactional Memory (TM), Thread-Level Speculation (TLS), and Checkpointed multiprocessors. In TM (e.g., [2, 12, 13, 18, 20]), the speculative threads are obtained from parallel programs, and the emphasis is typically on easing programmability. In TLS (e.g., [11, 15, 17, 22, 23, 24, 26, 27]), the speculative threads are extracted from a sequential program, and the goal is to speed-up the program. Finally, Checkpointed multiprocessors [5, 8, 14] provide primitives to enable aggressive thread speculation in a multiprocessor environment.

With the long-anticipated arrival of ubiquitous chip multiprocessor (CMP) architectures, it would appear that these techniques should have been architected into systems by now. The fact that they are not is, to some extent, the result of the conceptual and implementation complexity of these techniques.

Multiprocessor designs that support speculative multithreading need to address two broad functions: correctly maintaining the data dependences across threads and buffering speculative state. While the latter is arguably easier to understand (e.g., [9]), the former is composed of several complicated operations that typically involve distributed actions in a multiprocessor architecture — often tightly coupled with the cache coherence protocol. Specifically, this function includes mechanisms for: disambiguating the addresses accessed by different threads, invalidating stale state in caches, making the state of a committing thread visible to all other threads, discarding incorrect state when a thread is squashed, and managing the speculative state of multiple threads in a single processor.

The mechanisms that implement these five operations are hardware intensive and often distributed. In current designs, the first three piggy-back on the cache coherence protocol operations of the machine, while the last two typically modify the primary caches. Unfortunately, coherence protocols are complicated state machines and primary caches are delicate components. Modifications to these structures should minimize added complexity.

The goal of this paper is to simplify the conceptual and implementation complexity of these mechanisms. For that, we employ a Bloom-filter-based [3] compact representation of a thread's access information that we call a *Signature*. A signature uses hashing to encode the addresses accessed by a thread. It is, therefore, a superset representation of the original addresses. We also define a set of basic signature operations that efficiently operate on groups of addresses. These operations are conceptually simple and easy to implement in hardware. Finally, we use these operations as building blocks to enforce the data dependences across speculative threads and to correctly buffer speculative state.

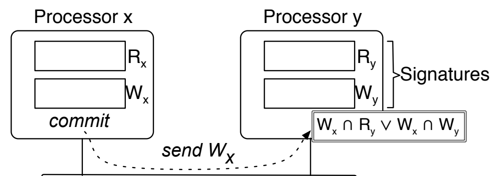

Figure 1. Example of an operation in Bulk.

Since signature operations operate on groups of addresses, we call our scheme *Bulk Disambiguation* or *Bulk*. Bulk operations are inexact — although correct execution is guaranteed. However, they are simple, as they often eliminate the need to record or operate on

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

individual addresses. As an example, Figure 1 shows two processors, each with its own signatures of addresses read (R) and written (W). As one thread commits, it sends its write signature to the other processor, where it is bulk-disambiguated against the signatures of the other thread. If no intersection is detected, there is no dependence violation. This is in contrast to conventional schemes, which have to disambiguate each address written by the first thread individually.

In this paper, we make two contributions. First, we introduce the concept and design of Bulk. Bulk is a novel approach to enforce data dependences across multiple speculative threads. The main characteristic of Bulk is that it operates on sets of addresses, providing substantial conceptual and implementation simplicity. Second, we evaluate Bulk in the context of both TLS using SPECint2000 codes and TM using multithreaded Java workloads. We show that, despite its simplicity, Bulk has competitive performance with more complex schemes. We also find that signature configuration is a key design parameter.

This paper is organized as follows: Section 2 is a brief background on speculative multithreading, Section 3 presents signatures and basic operations on them, Section 4 details the Bulk architecture, Section 5 demonstrates Bulk's simplicity, Section 6 describes implementation issues, Section 7 evaluates Bulk in the context of TLS and TM, and Section 8 presents related work.

2. Operations in Speculative Multithreading

Both TLS and TM are environments with multiple speculative threads. In TLS (e.g. [11, 15, 17, 22, 23, 24, 26, 27]), threads are tasks from a sequential program. Therefore, they need to appear to have executed in the same order as in the sequential execution. In TM (e.g., [2, 12, 13, 18, 20]), threads are typically obtained from a parallel program, and become speculative when they enter a transaction. While there is no predefined order of transactions, they have to appear to be atomic. In both TLS and TM, these thread ordering constraints impose an ordering of accesses to data across threads that, typically, the hardware has to enforce. As indicated in Section 1, enforcing these data dependences requires performing several operations. We briefly outline them here.

Disambiguating the Addresses Accessed by Different Threads. To ensure that data dependences required by thread ordering constraints are enforced, the hardware typically monitors the addresses accessed by each thread and checks that no two accesses to the same location may have occurred out of order. The process of comparing the addresses of two accesses from two different threads is called cross-thread address disambiguation. An access from a thread can be disambiguated *Eagerly* or *Lazily*. In Eager schemes, as soon as the access is performed, the coherence protocol propagates the request to other processors, where address comparison is performed. In Lazy schemes, the comparison occurs when the thread has completed and has broadcasted the addresses of all its accesses.

Making the State of a Committing Thread Visible to All Other Threads. While a thread is speculative, the state that it generates is typically kept buffered, and is made available to only a subset of the other threads (in TLS) or to no other thread (in TM). When the thread completes (and it is its turn in TLS), it commits. Committing informs the rest of the system that the state generated by the thread is now part of the safe program state. Committing often leverages the cache coherence protocol to propagate the thread's state to the rest of the system.

Discarding Incorrect State When a Thread Is Squashed. As addresses are disambiguated either eagerly or lazily, the hardware may find that a data dependence has been violated. In this case, the thread that is found to have potentially read or written a datum prematurely is squashed — in TLS, that thread's children are also squashed. When a thread is squashed, the state that it generated must be discarded. This involves accessing the cache tags and invalidating the thread's dirty lines or sometimes all the thread's lines.

Invalidating Stale State in Caches. Threads typically make their state visible at commit time. In addition, in some TLS systems, a thread can make its updates visible to its children threads immediately. In both cases, the cache coherence protocol of the machine ensures that the relevant caches in the system receive a coherence action — typically an invalidation for each updated line.

Managing the Speculative State of Multiple Threads in a Single Processor. A cache that can hold speculative state from multiple threads is called multi-versioned. Among other reasons, these caches are useful to be able to preempt and re-schedule a long-running TM transaction while keeping its state in the cache, or to avoid processor stall when TLS tasks are imbalanced. Specifically, in TLS, if tasks have load imbalance, a processor may finish a task and have to stall until the task becomes safe. If, instead, the cache is multi-versioned, it can retain the state of the old task and allow the processor to execute another task.

Multi-versioned caches are often implemented by extending the tag of each cache line with a version ID. This ID records which task the line belongs to.

Overall, implementing these operations requires significant hardware. Such hardware is often distributed and not very modular. It typically extends the cache coherence protocol or the primary caches — two hardware structures that are already fairly complicated or time-critical. The implementation of these operations is most likely the main contributor to the hardware complexity of speculative multithreading.

3. Signatures and Bulk Operations

To reduce the implementation complexity of the operations just described, this paper proposes a novel, simpler way of supporting them. Our goal is to simplify their hardware implementation while retaining competitive performance for the overall application.

The approach that we propose is called *Bulk* or *Bulk Disambiguation*. The idea is to operate on a group of addresses in a single, bulk operation. Bulk operations are relatively simple to implement, but at the expense of being inexact — although execution is always correct. This means that they may occasionally hurt performance but not correctness.

To support Bulk, we develop: (i) an efficient representation of sets of addresses and (ii) simple bulk operations that operate on such a representation. We discuss these issues next.

3.1. Address Signatures

We propose to represent a set of addresses as a *Signature*. A signature is generated by inexactly encoding the addresses into a

register of fixed size (e.g., 2 Kbits), following the principles of hash-encoding with allowable errors as described in [3]. Algebraically, given a set of addresses A_1, we use a hash function H such that $A_1 \longmapsto S$, where S is a signature. H is such that multiple sets of addresses can map into the same S. As a result of such aliasing, S encodes a superset of A_1. Consequently, when we later decode S with H^{-1} such that $S \longmapsto A_2$, we obtain a set of addresses A_2 such that: $A_1 \subseteq (A_2 = H^{-1}(H(A_1)))$.

Figure 2 illustrates how an address is added to a signature. The address bits are initially permuted. Then, in the resulting address, we select a few bit-fields $C_1, \ldots, C_n$. Each of these C_i bit-fields is then decoded and bit-wise OR'ed to the current value of the corresponding V_i bit-field in the signature. This operation is done in hardware.

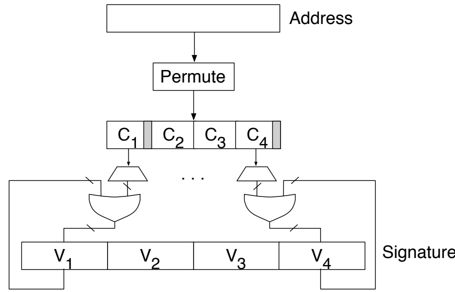

Figure 2. Adding an address to a signature.

Signature representation has aliasing. Our Bulk design is such that aliasing can hurt performance but not affect correctness. Moreover, Bulk builds the signatures to minimize performance penalties due to aliasing.

3.2. Primitive Bulk Operations

Bulk performs the primitive operations on signatures shown in Table 1. Signature intersection and union are bit-wise AND and OR operations, respectively, on two signatures. Intersecting two signatures produces a third signature that represents a superset of the addresses obtained by intersecting the original address sets. Specifically, for two sets A_1 and A_2, we have: $(A_1 \cap A_2) \subseteq H^{-1}(H(A_1) \cap H(A_2))$. A similar effect occurs for unions.

Op.	Description	Sample Use
$\cap$	Signature intersection	Address disambiguation
$\cup$	Signature union	Combining write signatures in transaction nesting
$= \emptyset$	Is signature empty?	Address disambiguation
$\in$	Membership of an address in a signature	Address disambiguation with individual address
δ	Signature decoding into sets (exact)	Signature expansion

Table 1. Primitive bulk operations on signatures.

Checking if a signature is empty involves checking if at least one of its V_i bit-fields is zero. If so, the signature does not contain any address. The membership operation ($\in$) checks if an address a can be in a signature S. It involves encoding a into an empty signature as discussed in Section 3.1, then intersecting it with S, and finally checking if the resulting signature is empty.

Ideally, we would like to be able to decode a signature into its contributing addresses A_1. However, this is potentially time consuming and can generate only a superset of the correct addresses. Instead, we define the decode operation (δ) to generate the *exact* set of *cache set indices* of addresses A_1. We will see that this operation is useful in cache operations using signatures. It can be implemented easily by simply selecting one of the C_i bit-fields to be the cache index bits of the address and, therefore, the corresponding V_i will be the cache set bitmask. This particular implementation is not required — if the index bits of the address are spread over multiple C_i, the cache set bitmask can still be produced by simple logic on multiple V_i.

Table 1 also lists a sample use of each operation. We will discuss the uses in Section 4. Finally, Figure 3 shows how these operations are implemented.

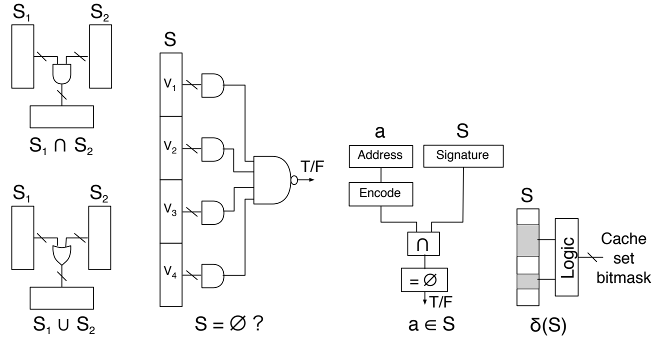

Figure 3. Implementation of the primitive bulk operations on signatures.

3.3. Signature Expansion

There is one other important Bulk operation that is composed of two of the primitive operations in Table 1. This operation is called *Signature Expansion*, and it involves determining which lines in the cache may belong to a signature. This operation is defined as $H^{-1}(S) \cap T$, where S is the signature being expanded and T is the set of line addresses present in the cache.

A naive implementation would simply walk the cache tags, take every line address that is valid, and apply the membership operation to it. Unfortunately, this is very inefficient, since the number of matching line addresses may be small. Instead, we can use the decoding operation δ on the signature to obtain the cache set bitmask. Then, for each of the selected sets, we can read the addresses of the valid lines in the set and apply the membership operation $\in$ to each of them.

Figure 4 shows the implementation of signature expansion. The result of applying δ on a signature is fed to a finite state machine (FSM). The FSM then generates, one at a time, the index of the selected sets in the bitmask. As each index is provided to the cache, the cache reads out all the valid line addresses in the set. These addresses are then checked for membership in the signature.

4. An Architecture for Bulk Disambiguation

Based on these primitive operations, we can now build the complete *Bulk* architecture. Bulk presumes a multiprocessor with an invalidation-based cache coherence protocol. An application can run both non-speculative and speculative threads. The former send

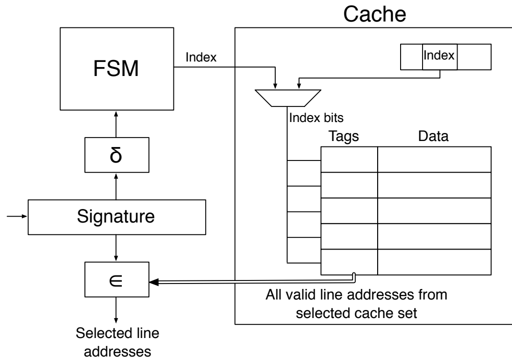

Figure 4. Implementation of signature expansion.

invalidations as they update lines; the latter do not send any invalidations until they attempt to commit. At that point, they send a single message out to inform the other threads of a superset of the addresses that they have updated — without sending out the full set of addresses or the data that they have generated. Based on this message, other threads may get squashed and/or may invalidate some of their cache lines. Bulk is, therefore, a lazy scheme as described in Section 2.

In Bulk, every speculative thread has a *Read* (R) and a *Write* (W) signature in hardware (Figure 1). At every load or store, the hardware adds the requested address to R or W, respectively, as shown in Figure 2. If the speculative thread is preempted from execution, its R and W signatures are still kept in the processor.

In the following, we describe the operation of Bulk, including thread commit and squash, bulk address disambiguation, bulk invalidation, and disambiguation at fine grain. We conclude with the overall architecture of Bulk.

4.1. Thread Commit and Squash

Consider a speculative thread *C* that finishes and wants to commit its speculative state. It first obtains permission to commit (e.g. gaining ownership of the bus). When the thread knows that its commit will proceed, it clears its W_C and R_C signatures. Then, it sends out its write signature W_C so that it can be disambiguated against all other threads in the system (Figure 1). This is shown in Figure 5(a).

In Bulk, the committing thread never sends the expanded list of addresses it wrote. Moreover, note that Bulk is not concerned about how the system handles commit races — several threads attempting to commit at once. This is a matter for the protocol and network to support. However, by sending only a single signature message, Bulk may simplify the handling of such races.

Figure 5(b) shows the actions at a thread *R* that receives the signature from the committing one. First, it performs *Bulk Address Disambiguation* (Section 4.2) against its local read (R_R) and write (W_R) signatures. This operation decides whether the thread needs to be squashed. If it is, thread *R* uses its write signature (W_R) to *Bulk Invalidate* (Section 4.3) all the cache lines that it speculatively modified[1]. Then, it clears its R_R and W_R.

Regardless of the outcome of the bulk address disambiguation, all the lines written by the committing thread that are present in thread *R*'s cache need to be invalidated. This is done by using the

[1] In TLS, other cache lines may be invalidated as well (Section 6.3).

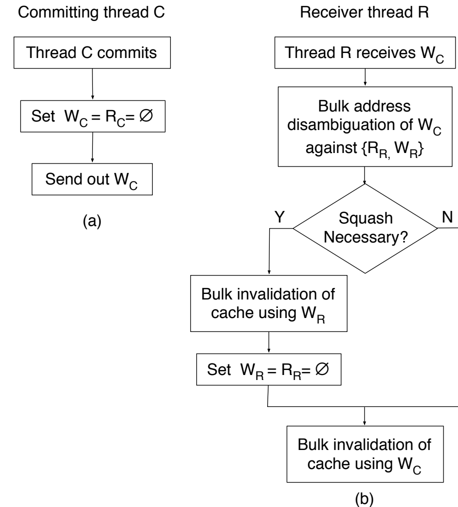

Figure 5. Flowchart of the commit process: committing thread (a) and receiver thread (b).

write signature of the committing thread (W_C) to perform a bulk invalidation on thread *R*'s cache.

4.2. Bulk Address Disambiguation

The memory addresses written by a committing thread *C* are disambiguated in hardware against the memory addresses accessed by a receiver thread *R* using bulk operations on signatures. If

$$W_C \cap R_R \neq \emptyset \quad \vee \quad W_C \cap W_R \neq \emptyset \qquad (1)$$

then we have detected a potential read-after-write or a potential write-after-write dependence between the threads. In this case, thread *R* is squashed; otherwise, it may continue executing. Write-after-write dependences induce squashes because threads could have updated a fraction of a line, and because an additional reason discussed in Section 4.4.

Bulk disambiguation is very fast and simple. However, it may have false positives due to address aliasing and cause unnecessary squashes. In our experiments of Section 7, we show that the number of false positives is reasonable and does not affect performance significantly. We also show that signatures can be constructed to minimize the number of false positives.

Signatures are designed to encode a certain granularity of addresses — e.g., line addresses or word addresses. In each case, disambiguation occurs at the granularity encoded in the signature. However, if we disambiguate at a granularity smaller than the cache line, the hardware has to be able to merge partial updates of lines. Section 4.4 discusses this issue.

Finally, not all disambiguations are done in bulk. Non-speculative threads send individual invalidations as they update lines. In this case, when *R* receives an invalidation for address a, it uses the membership operation to check if $a \in R_R \vee a \in W_R$. If the test is true, *R* is squashed.

4.3. Bulk Invalidation

A thread R performs bulk invalidation in two cases. The first one is when it is squashed; it uses its W_R to invalidate all its dirty

cache lines. The second one is when it receives the write signature of a committing thread (W_C); it invalidates all the lines in its cache that are in W_C.

In Bulk, the first case would not work correctly if a cached dirty line that is either non-speculative or was written by another speculative thread S appears, due to aliasing, to belong to W_R. Bulk would incorrectly invalidate the line.

To avoid this problem while still keeping the hardware simple, Bulk builds signatures in a special way, and restricts in a certain way the dirty lines that can be in the cache at a time. Specifically, Bulk builds signatures so that the decode operation $\delta(W)$ of Section 3.2 can generate the *exact* set of cache set indices of the lines in W. Section 3.2 discussed how this is easily done. In addition, Bulk enforces that any dirty lines in a given cache set can only belong to a single speculative thread or be non-speculative. In other words, if a cache set contains a dirty line belonging to speculative thread S, any other dirty line in that same set has to belong to S — although no restrictions are placed on non-dirty lines. Similarly, if a cache set contains a non-speculative dirty line, any other dirty line in the set has to be non-speculative as well. We call this restriction the *Set Restriction*. Section 4.5 explains how Bulk enforces the Set Restriction. Overall, with the way Bulk generates signatures and the Set Restriction, we have solved the problem — Bulk will not be incorrectly invalidating dirty lines.

We can now describe how the two cases of bulk invalidation proceed. They start by performing Signature Expansion on the write signature (W_R for the first case and W_C for the second one). Recall from Section 3.3 that Signature Expansion is an operation composed of two primitive Bulk operations. It finds all the lines in the cache that may belong to W. It involves applying $\delta(W)$ and, for each of the resulting sets, reading all the line addresses a and applying the membership test $a \in W$. For each address b that passes the membership test, the two cases of bulk invalidation perform different operations.

In the case of invalidating dirty lines on a squash, Bulk checks if b is dirty. If so, Bulk invalidates it. Thanks to the way signatures are built and the Set Restriction, b cannot be a dirty line that belongs to another speculative thread or is non-speculative.

In the case of invalidating the addresses present in the write signature of a committing thread C, Bulk checks if b is clean. If so, Bulk invalidates it. It is possible that b passed the membership test due to aliasing and therefore is not really in W_C. If so, we may hurt performance but not correctness. In addition, note that Bulk takes no action if b is dirty. The reason is that this is the case of a non-speculative dirty line whose address appears in W_C due to aliasing. Indeed, if b belonged to a speculative thread, it would have caused a squash. Moreover, it cannot be dirty non-speculative and be written by the committing thread C.

4.4. Disambiguating at Fine Grain

If the signatures are built using addresses that are of a finer granularity than memory lines, then the bulk disambiguation occurs at that granularity. For example, if signatures use word addresses, two speculative threads that have updated different words of a line will commit without causing a dependence violation (except if aliasing occurs). Word-level disambiguation improves performance in many TLS codes [6], but requires that the partially updated memory lines merge in the order in which the threads commit. Bulk supports this case without modifying the cache or the cache coherence protocol.

To do so, Bulk slightly modifies the process of bulk invalidation for the case when it needs to invalidate the lines that are in the write signature W_C of a committing thread C. Specifically, consider that the committing thread C and a second thread R have written to a different word of a line. Since we encode word addresses in the signatures, when R performs the bulk disambiguation of the arriving W_C against its own W_R and R_R, it finds no violation. However, as it performs bulk invalidation, it can find a cache line whose address b passes the membership test, is dirty, and (this is the new clue) is in a cache set present in $\delta(W_R)$. This line has suffered updates from both threads.

In this case, Bulk has to merge the two updates and keep the resulting line in R's cache. To do so, Bulk uses W_R and b to generate a (conservative) bitmask of the words in the line that were updated by R. This is done with an Updated Word Bitmask functional unit that takes and manipulates the appropriate bits from W_R (Figure 6). This bitmask is conservative because of word-address aliasing. However, it cannot include words that were updated by the committing thread C — otherwise, R would have been squashed in the disambiguation operation. Then, Bulk reads the line from the network and obtains the version just committed. The committed version is then updated with the local updates specified in the word bitmask (Figure 6), and the resulting line is written to the cache. Note that this process requires no cache modifications — not even per-word access bits.

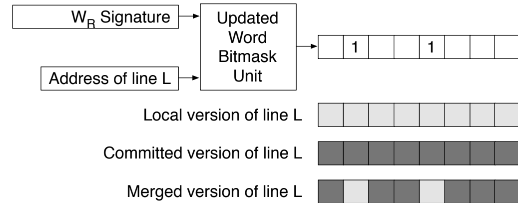

Figure 6. Merging lines partially updated by two speculative threads.

From this discussion, it can be deduced why the $W_C \cap W_R$ component of Equation 1 is required even in word-level disambiguation. Specifically, W_R is conservative due to aliasing, and the word bitmask of Figure 6 could include (due to aliasing) words that thread C wrote. In this case, if Bulk did not perform the $W_C \cap W_R \neq \emptyset$ test and did not squash R, we would be incorrectly merging the lines.

4.5. Overall Bulk Architecture

Based on the previous discussion, Figure 7 shows the overall Bulk architecture. It is placed in a *Bulk Disambiguation Module* (BDM) that includes several components. The BDM has a read and a write signature for each of the several speculative versions supported by the processor. Supporting multiple speculative versions is useful for buffering the state of multiple threads or multiple checkpoints.

The BDM also has a set of functional units. They perform the primitive bulk operations of Table 1, the Signature Expansion of Section 3.3, and the bitmask of updated words of Section 4.4.

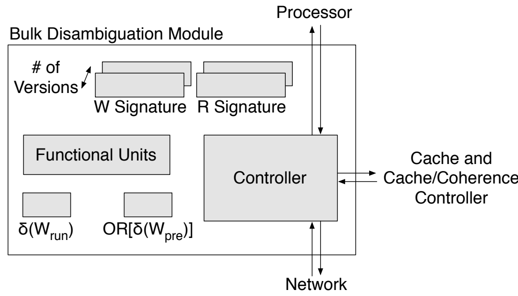

Figure 7. Overview of the Bulk Disambiguation Module (BDM).

The BDM includes two registers with as many bits as sets in the cache. They contain cache set bitmasks resulting from applying the decode operation (δ) to certain signatures. Specifically, one decodes the write signature of the thread that is currently running on the processor ($\delta(W_{run})$). The other contains the logical-OR of the decoded versions of all the other write signatures in the processor. They belong to speculative threads that have state in the cache but have been preempted from the CPU ($OR(\delta(W_{pre}))$). This bitmask is updated at every context switch.

These bitmasks are used to identify which dirty lines in the cache are speculative and to which thread they belong. This is necessary because the cache has no notion of what lines or words are speculative — we keep the cache *unmodified* relative to a non-speculative system. For example, consider an external read request that reaches the BDM and wants to access a cache set that has a bit set in the $\delta(W_{run})$. We know that any dirty line in that set is speculative and belongs to the running thread. Consequently, the BDM nacks the request, preventing it from getting speculative data. If the request wanted to read a line that was clean in the cache, no harm is done, since the memory will respond.

In addition, these bitmasks also help the BDM Controller maintain the Set Restriction. Specifically, when a speculative thread issues a write to the cache, the BDM Controller checks the local $\delta(W_{run})$ and $OR(\delta(W_{pre}))$. If both bitmasks have a zero in the entry corresponding to the requested set, the write can proceed to the cache. However, before the write is allowed to update the cache, any dirty line in the corresponding cache set is written back to memory. The corresponding entry in $\delta(W_{run})$ is then set.

If, instead, the bitmask entries are (1,0), respectively, the write can update the cache directly. Finally, if they are (0,1), a special action is taken to preserve the Set Restriction. Depending on the implementation, this can be preempting the thread, squashing the preempted thread that owns the dirty lines in the set, or merging the two threads that want to own lines in the same set. Overall, with this support, Bulk guarantees the Set Restriction. Thanks to this restriction and the way Bulk builds signatures, it is the case that, for any two write signatures W_1 and W_2 in the processor, $W_1 \cap W_2 = \emptyset$.

5. Simplicity of Bulk Disambiguation

A key feature of Bulk is its conceptual and implementation simplicity. Its simplicity is due to two reasons: its compact representation of sets of addresses, and its definition of a collection of basic primitive operations on the sets. We discuss each reason in turn.

5.1. Compact Representation of Address Sets

Bulk represents the set of addresses accessed speculatively very concisely. This simplifies the hardware implementation of several operations. Table 2 lists some of the key simplifications in Bulk. We discuss each one in turn.

Key simplifications
Send only a write signature at commit
Single-operation full address disambiguation
Inexpensive recording of speculatively-accessed addresses
Compact version representation without version IDs
Fine-grain (per word) disambiguation with no extra storage
Commit by clearing a signature

Table 2. Key simplifications in Bulk.

A committing thread in Bulk only sends a short, fixed-sized message with its write signature (e.g., 2 Kbits) to all other threads. It does not broadcast the list of individual addresses. This enables more efficient communication and perhaps simpler commit arbitration. Perhaps more importantly, Bulk does not need to walk the cache tags to collect any addresses to broadcast, nor to buffer them before sending them. These issues complicate the commit implementation in conventional systems.

Bulk disambiguates all the addresses of two speculative threads in one single operation. While false positives are possible, our experiments suggest that their frequency is tolerable. In conventional lazy systems, disambiguation is typically lengthy and complicated, as each individual address is checked against the cache tags. Conventional eager systems disambiguate each write separately.

Bulk records the speculatively-read and written addresses in an R and a W signature, avoiding the need to modify the tags of cache lines with a Speculative bit. Moreover, consider a long-running thread that reads many lines. Read-only lines can be evicted from the cache in both Bulk and conventional systems, but the system must record their addresses for later disambiguation. Conventional systems require a special hardware structure that grows with the thread length to record (and later disambiguate) all these evicted addresses. Bulk simply uses the R signature. Written lines that are evicted are handled in a special manner in both conventional systems and Bulk (Section 6.2.2).

Bulk represents multi-version information very concisely. For each version or checkpoint, Bulk stores a read and a write signature. Another related Bulk structure is a cache set bitmask generated from the W of all the preempted threads (Figure 7). In contrast, conventional systems typically tag each cache line with a version ID, whose size depends on the number of versions supported. Setting, managing, comparing, and clearing many version IDs introduces significant hardware complexity.

Bulk can build signatures using fine-grain addresses (e.g., word or byte) and therefore enable fine-grain address disambiguation without any additional storage cost. In contrast, conventional schemes that perform fine-grain disambiguation typically add per-word read and write access bits to each cache line. These bits add significant complexity to a structure that is time-critical.

Finally, Bulk commits a thread by clearing its read and write signatures. This is a very simple operation, and is not affected by the

number of versions in the cache. In contrast, conventional schemes either gang-clear a Speculative bit in the cache tags, or walk the cache tags to identify the lines belonging to the committing thread. Neither approach is simple to implement, especially when there are lines from many threads in the cache.

5.2. Basic Primitive Operations

The second reason for Bulk's simplicity is that is uses a set of well-defined basic operations. They are those in Table 1, the Signature Expansion of Section 3.3, and the Updated Word Bitmask operation of Section 4.4. These operations map high-level computations on sets directly into hardware.

6. Implementation Issues

This section examines some implementation details relevant to signatures and the use of Bulk in TM and TLS.

6.1. Signature Encoding

Address aliasing in signatures is an important concern for Bulk because it may degrade performance. Therefore, we would like to choose a signature implementation that minimizes aliasing. Given our generic signature mechanism presented in Section 3.1, there are many variables that can be adjusted to control address aliasing, including the total size of the signature, the number and size of the V_i and C_i bit-fields, and how the address bits are permuted. The whole space is very large but the main trade-off is between signature size and accuracy — the latter measured as the relative absence of false positives. We evaluate this trade-off in Section 7.5.

Although signatures are small (e.g., 2 Kbits), we may want to further reduce the cost of sending them over the interconnection network. Specifically, since they potentially have many sequences of zeros, we compress them with run-length encoding (RLE) before broadcasting. RLE is simple enough to be easily implemented in hardware and is highly effective at compressing signatures. We analyze RLE's impact in Section 7.5.

6.2. Issues in Transactional Memory (TM)

Two important issues in TM are transaction nesting and the actions taken on cache overflow and context switch. We consider how Bulk addresses them.

6.2.1. Transaction Nesting

Figure 8 shows a transaction nested inside another. The transaction begin and end statements divide the code into three sections, labeled *1*, *2*, and *3*. An intuitive model of execution for this code is that of closed nested transactions with partial rollback. In closed nested transactions [19], an inner transaction does not become visible to the other threads until the outer transaction commits. With partial rollback, we mean that, if a dependence violation is detected on an access in section *i*, we only rollback execution to the beginning of section *i* and re-execute from there on.

Bulk can easily support this model. Recall from Section 4.5 that a processor's BDM supports multiple versions, each one with a read and a write signature. Consequently, Bulk creates a separate read and write signature for each of the three code sections in the figure. We call them R_1 and W_1, R_2 and W_2, and R_3 and W_3 in the figure.

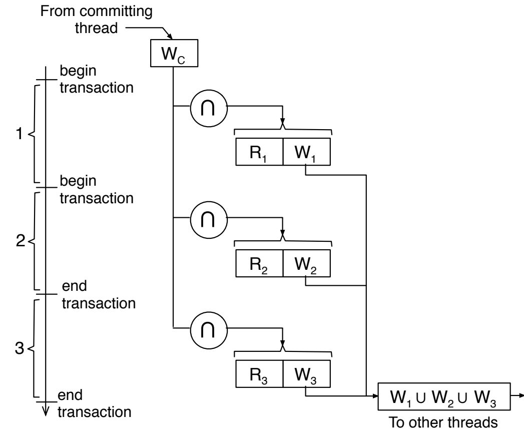

Figure 8. Supporting nested transactions in Bulk.

As shown in the figure, when the thread receives a W_C from a committing transaction, it performs bulk address disambiguation in order, starting from R_1 and W_1, and finishing with R_3 and W_3. If a violation is detected in a section (e.g., in *3*), only that section and the subsequent ones are squashed and restarted.

The three pairs of signatures are kept until section *3* finishes execution. At that point the outer transaction attempts to commit. The write signature that it broadcasts to all other threads is the union of W_1, W_2, and W_3.

6.2.2. Overflow and Context Switch

In TM, two potentially costly events are the overflow of speculative lines from the cache and the preemption of an executing speculative thread in a context switch. In the former, conventional schemes typically send the overflowed line addresses (and in the case of dirty lines their data as well) to an overflow area in memory, where the addresses still need to be checked for potential dependences [2, 20]. In a context switch, many conventional schemes move the cached state of the preempted speculative thread to the overflow area [2, 20].

Bulk reduces the complexity and the performance overhead of having to deal with overflows and context switches. The three reasons are shown in Table 3. In the following, we consider overflow and context switches in turn.

Addresses of overflowed lines are not accessed when disambiguating threads
A processor efficiently determines if it needs to access the overflow area
Supporting multiple R and W signatures in the BDM substantially minimizes overheads

Table 3. Ways in which Bulk reduces the complexity and performance overhead of overflows and context switches.

In Bulk, when dirty lines from a speculative thread are evicted from the cache, they are moved to a per-thread overflow area in memory. However, recall that address disambiguation in Bulk is exclusively performed using signatures. Therefore, unlike in conventional schemes, the overflowed addresses in memory are not accessed when Bulk disambiguates threads. A thread with overflowed

lines that receives a W_C from a committing thread simply operates on its signatures and performs bulk invalidation of cached data only. It only accesses its overflow area to deallocate it — if the disambiguation found a dependence.

During a speculative thread's normal execution, a thread may request data that happens to be in its overflow area. Bulk provides an efficient mechanism for the processor to know whether it needs to access the overflow area. Specifically, when a thread overflows, the BDM sets an Overflow bit (O) for it. When the thread next misses in the cache (say, on address a), as the BDM intercepts the request (Figure 7), it checks the O bit. If it is set, the BDM tests if $a \in W$. If the result is false, the request does not need to access the overflow area, and can be sent to the network.

Finally, consider context switches. In Bulk, when a thread is preempted, it still keeps its R and W signatures in the BDM. A new pair of signatures is assigned to the newly scheduled thread (Section 4.5). Consequently, as long as there are enough R and W signatures in the processor's BDM for the running and preempted threads, disambiguation proceeds as efficiently as usual in the presence of overflows and context switches.

When a processor finally runs out of signatures, the R and W signatures of one thread, say i, are moved to memory. In this case, the thread's cached dirty lines are also moved to memory — since the cache would not know what thread is the owner of these dirty lines. From here on, the W_C of committing threads and individual writes from non-speculative threads need to disambiguate against the R_i and W_i in memory. This operation is similar to the disambiguation against overflowed addresses in memory that is supported in conventional systems (e.g., [20]) — yet simpler, because signatures are small and fixed-sized, while overflowed line addresses need a variable amount of storage space. When space opens up in the BDM, the R_i and W_i signatures are reloaded, possibly together with some or all its dirty lines in the overflow area.

6.3. Issues in Thread-Level Speculation (TLS)

As far as Bulk is concerned, the key difference between TLS and TM is that, in TLS, speculative threads can read speculative data generated by other threads. As a result, Bulk needs to be extended slightly. Specifically, when a thread is squashed, it also uses its R signature to bulk-invalidate all the cache lines that it has read. The reason is that they may contain incorrect data read from a predecessor speculative thread that is also being squashed.

We also note that threads in TLS often have fine-grain sharing, especially between a parent thread and the child thread that it spawns. The child often reads its live-ins from the parent shortly after being spawned. If we keep Bulk as is, the child will often be squashed when the parent commits.

To enhance performance, we propose one improvement: not to squash a thread if it reads data that its parent generated before spawning it. We call this improvement *Partial Overlap*. For simplicity, we only support it for the first child of a given thread.

Partial Overlap requires three extensions to Bulk. The first one is that, at the point where a thread spawns its first child, the hardware starts generating a shadow write signature W_{sh} in parallel as it builds up the usual write signature W (Figure 9). From the point of the spawn on, both signatures are updated at every write.

Secondly, when a thread commits, it sends both its write signature W and its shadow one W_{sh} (Figure 9). Its first child — identified by its thread ID — uses W_{sh} for disambiguation, while all other threads use W. Finally, when a thread spawns it first child, it passes along with the spawn command its current W (Figure 9). In the receiving processor, the spawn operation includes a bulk invalidation of the clean cached lines whose addresses are in W.

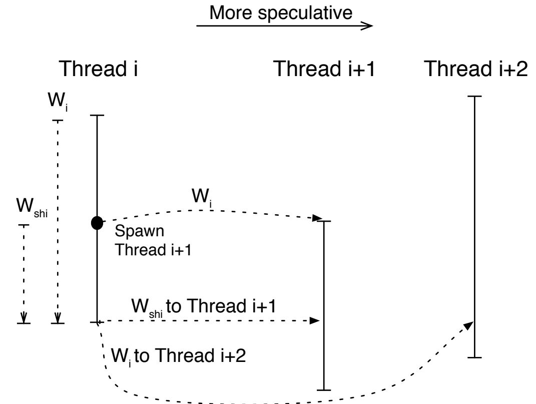

Figure 9. Supporting Partial Overlap in TLS.

With this support, before the child thread starts in a processor, the cache is emptied of any addresses whose data has been modified by the parent. Consequently, on accessing such addresses, the child will miss in its cache and obtain the data from its parent's. Then, when the parent commits, address disambiguation will not include addresses updated by the parent only before spawning the child.

7. Evaluation

In this section, we evaluate Bulk in the context of both TM and TLS. After presenting our evaluation setup in Section 7.1, we show that Bulk induces a very small performance degradation in Section 7.2. We characterize the main aspects of the bulk operations in Section 7.3. We then present bandwidth issues in Section 7.4. Finally, we show the trade-offs of signature encoding in Section 7.5.

7.1. Evaluation Setup

For the TLS evaluation, we compile the applications using a fully automatic profile-based TLS compilation infrastructure [16]. We run the binaries on an execution-driven simulator [21] with detailed processor core and memory system models, including all TLS operation overheads, such as thread spawn, thread squash, and versioning support. We used the SPECint2000 applications running the *ref* data set. We run all the SPECint2000 applications except *eon* (C++ is not supported) and *gcc* and *perlbmk* (our compiler cannot handle them).

For the TM evaluation, we modified Jikes RVM [1] to add *begin* and *end* transaction annotations to Java programs. We convert lock-based constructs into transactions using methods similar to [4]. We ran our modified Jikes RVM on the Simics full-system simulator enhanced to collect memory traces and transaction annotations. These traces were then analyzed in our TM simulator. Our TM model supports execution of non-transactional code as in [2, 18, 20]. The TM simulation includes a detailed memory model. As listed in Table 4, the applications used were SPECjbb2000 and programs from the Java Grande Forum (JGF) multithreaded benchmarks package.

Application	Suite	Description
cb	JGF	Cryptography Benchmark
jgrt	JGF	3D Ray Tracer
lu	JGF	LU matrix factorization
mc	JGF	Monte-Carlo simulation
moldyn	JGF	Molecular dynamics
series	JGF	Fourier coefficient analysis
sjbb2k	SPEC	SPECjbb 2000 (business logic)

Table 4. Java applications used in the TM experiments.

Table 5 presents the architectural parameters for the TLS and TM architectures. For the TM experiments, the signatures are configured to encode line addresses. For TLS, since the applications evaluated have fine-grain sharing, signatures are configured to encode word addresses. In both the TLS and TM experiments, we compare Bulk to conventional systems that perform exact address disambiguation. Conventional systems can be *Eager*, if disambiguation occurs as writes are performed, or *Lazy*, if threads disambiguate all their updates when they commit. Finally, our baseline Bulk includes support for Partial Overlap in TLS (Section 6.3) and for overflows and context switches in TM (Section 6.2.2). It does not include support for partial rollback of nested transactions (Section 6.2.1).

TLS	
Processors	4
Fetch, issue, retire width	4, 3, 3
ROB, I-window size	126, 68
LD, ST queue entries	48, 42
Mem, int, fp units	2, 3, 1
L1 cache:	
size, assoc, line	16 KB, 4, 64 B
OC, RT	1, 2 cycles
RT to neighbor's L1 (min)	8 cycles
TM	
Processors	8
L1 cache:	
size, assoc, line	32 KB, 4, 64 B
Signature Information (Both TLS and TM)	
Default signature: *S14* (2 Kbits long, see Table 8 for details)	
Bit permutations used: (bit indices, LSB is 0)	
TM: [0-6, 9, 11, 17, 7-8, 10, 12, 13, 15-16, 18-20, 14]	
TLS: [0-9, 11-19, 21, 10, 20, 22]	

Table 5. Architectural parameters used in the simulations. OC and RT stand for occupancy and round trip from the processor, respectively. In the permutations, the bit indices are from line addresses (26 bits) in TM and from word addresses (30 bits) in TLS. The high-order bits not shown in the permutation stay in their original position.

7.2. Performance

Figure 10 shows the performance of Eager, Lazy, and Bulk in TLS compared to sequential execution. The results show that using Bulk has little impact on performance — only a geometric mean slowdown of 5% over Eager. Most of the performance degradation happens when going from Eager to Lazy. This degradation comes mainly from not restarting offending tasks as early as Eager does. The small difference between Lazy and Bulk is due to the inexactness of signatures.

Figure 10 also includes the performance of Bulk without the support for Partial Overlap discussed in Section 6.3. This is shown in the BulkNoOverlap bar. The geometric mean speedup of BulkNoOverlap is 17% lower than that of Bulk. The reason for this significant difference is that SPECint applications tend to have fine-grain sharing, especially between adjacent threads. As a related point, in these experiments, Lazy also includes support for a scheme similar to Partial Overlap but with exact information. We do this to have a fair comparison with Bulk.

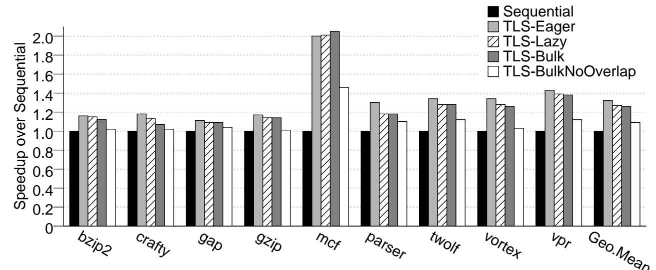

Figure 10. Performance of Eager, Lazy, and Bulk in TLS.

Figure 11 compares the performance of Bulk, Lazy, and Eager in the context of TM. The performance of Bulk and Lazy is approximately the same. We expected to see similar performance in Eager and Lazy, which is the case for all applications except SPECjbb2000. There are two reasons why SPECjbb2000 is faster in Lazy than in Eager. First, there is a situation where Eager has forward progress problems. This is shown in Figure 12(a), where two threads read and write the same location inside a transaction, and they keep squashing each other repeatedly[2]. The second reason involves a situation where a squash happens in Eager but not in Lazy, as shown in Figure 12(b).

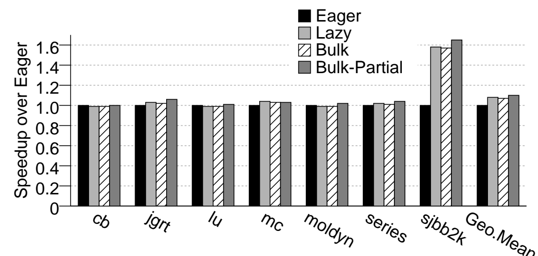

Figure 11. Performance of the different schemes in TM.

The Bulk-Partial bar in Figure 11 shows the performance of supporting partial rollback in nested transactions (Section 6.2.1). The plot shows that the impact of partial rollback is minor. This is due to the relatively low frequency and depth of transaction nesting in our Java applications. In addition, we observed that nested transactions frequently access common pieces of data, which makes it likely that if a conflict happens, it will involve multiple sections of a nested transaction, decreasing the benefits of partial rollback.

[2] To solve this problem in Eager, we detect this situation and choose to let the longer-running thread make progress and commit, while the other thread stalls.

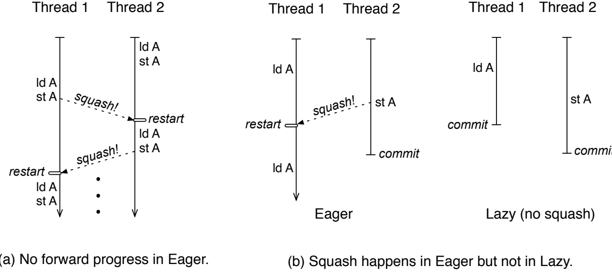

Figure 12. Examples of code patterns from SPECjbb2000 where the performance of Eager suffers.

7.3. Characterization of Bulk

Table 6 characterizes Bulk in TLS. The columns labeled *Task Properties* show the average sizes in words of the read and write sets (i.e., footprints) of the tasks. They also show the average size in words of the dependence sets of the squashed tasks. The dependence set is the result of the intersection between a committing task's write set and the read and write sets of the squashed task. Note that read sets tend to be significantly larger than write sets. Also, dependence sets are small.

Appl	Task Properties			False Positives		Set Restriction	
	Rd Set Size (W)	Wr Set Size (W)	Dep Set Size (W)	Sq (%)	False Inv/Com (Avg)	Safe WB/Tsk (Avg)	Wr-Wr Cnf/1k Tasks (Avg)
bzip2	30.2	4.9	1.0	10.5	0.1	2.9	0.1
crafty	109.0	23.2	2.6	16.5	0.0	11.5	0.3
gap	42.4	13.4	6.6	0.4	0.5	3.7	0.0
gzip	14.3	4.8	2.0	1.4	0.0	1.5	0.0
mcf	12.3	0.7	1.0	1.1	0.0	0.4	0.0
parser	29.6	7.1	2.3	2.1	0.1	2.2	5.5
twolf	41.1	6.4	1.4	14.0	0.3	6.3	0.2
vortex	34.7	23.5	3.6	10.4	0.3	6.4	31.6
vpr	43.1	8.7	1.1	5.6	0.5	4.1	0.0
Avg	39.6	10.3	2.4	6.9	0.2	4.3	4.2

Table 6. Characterization of Bulk in TLS.

The columns labeled *False Positives* characterize the impact of address aliasing in signatures. The *Squash* column shows the percentage of task squashes that were caused by collisions between signatures due to aliasing. The *False invalidations per commit* column shows the average number of cache lines that were invalidated due to aliasing during a bulk invalidation following a task commit. The figure shows the total over all the caches for a single commit operation. Overall, these numbers are low and explain why false positives do not affect performance much in Figure 10.

The columns labeled *Set Restriction* show the impact of using our Set Restriction (Section 4.3). The *Safe WB per task* column shows how many non-speculative dirty lines had to be written back to memory per task due to the Set Restriction. These lines often remain in the cache in clean state, since the victim line is another line in the set. The *Wr-Wr conflicts per 1000 tasks* column shows how often a speculative task attempts to write a line in a set that already contains a dirty line from another speculative task. In these cases, the most speculative task of the two is squashed to keep the Set Restriction. We see that this situation is very infrequent, happening on average only 4 times every 1000 tasks.

Table 7 characterizes Bulk in the context of TM. The *Transaction Properties* columns show the average read and write set sizes of the transactions, and the dependence set size of the squashed transactions. The set sizes are measured in line lines. As expected, read set sizes are always a few times larger than write set sizes. On average, write sets hold 22 lines, while read sets hold 68 lines. Enumerating the addresses in hardware would induce considerable overhead. On average, the size of the dependence set is only about 2 lines.

Appl	Transaction Properties			False Positives		Set Rest.	Overflow
	Rd Set Size (L)	Wr Set Size (L)	Dep Set Size (L)	Sq (%)	False Inv/Com (Avg)	Safe WB/Tr (Avg)	Accesses Bulk/Lazy (%)
cb	73.6	26.9	1.4	20.0	0.6	1.5	6.2
jgrt	67.1	22.1	1.3	22.1	0.2	0.5	4.3
lu	81.7	27.3	1.3	12.8	0.7	0.8	5.6
mc	51.6	17.6	1.9	9.8	0.1	2.6	3.3
moldyn	70.2	25.1	1.3	10.7	0.4	0.4	2.6
series	86.9	25.9	1.1	13.7	0.1	0.3	2.1
sjbb2k	41.6	11.2	1.4	7.7	0.1	0.2	0.8
Avg	67.5	22.3	1.4	13.8	0.3	0.9	3.6

Table 7. Characterization of Bulk in TM.

The *False Positives* columns show information similar to the corresponding columns of Table 6. They show that on average 14% of the squashes are caused by signature collisions due to aliasing, and that the number of lines invalidated at commit due to aliasing is only 3 lines every 10 transaction commits.

The *Set Restriction* column shows that, on average, less than one non-speculative dirty line has to be written back to memory per transaction due to the Set Restriction. Finally, the *Overflow* column compares the number of accesses to the overflow area in Bulk and Lazy. Specifically, the column shows the number of such accesses in Bulk as a fraction of those in Lazy. We see that, on average, Bulk accesses the overflow area only 4% of the times that Lazy does. The savings come from the fact that Bulk does not access the overflow area on address disambiguation and that Bulk can use a membership operation to decide that an access to the overflow area is not necessary (Section 6.2.2). We can see that Bulk is very effective at avoiding accesses to the overflow area.

7.4. Bandwidth Usage in TM

We study the bandwidth usage in TM by looking at both the total bandwidth and the commit bandwidth usage. Figure 13 shows the breakdown of the total bandwidth used. We compare Eager (E), Lazy (L), and Bulk (B). The bandwidth is broken down into: invalidations (*Inv*), coherence messages like downgrades and upgrades (*Coh*), accesses to the unbounded memory area that holds overflowed data (*UB*), writebacks (*WB*), and line fills (*Fill*).

The figure shows that, despite the inaccuracy introduced by address aliasing, the overall bandwidth usage in Bulk is along the lines of that in the other schemes. On average, it is only slightly higher than Lazy and is lower than Eager. Bulk's bandwidth is higher than Lazy's because it has more line fills. They are due to the extra squashes and line invalidations caused by address aliasing.

Most of the *Inv* bandwidth usage in Lazy and Bulk is due to the commit operations (since individual invalidations from non-speculative threads are few). While it is hard to see in the figure, Bulk significantly reduces the commit bandwidth. The reason is

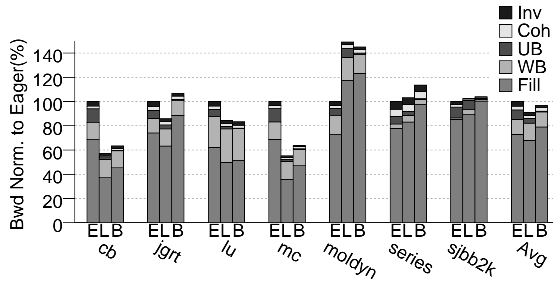

Figure 13. Bandwidth usage breakdown in TM. Underneath the bars, *E*, *L* and *B* refer to Eager, Lazy and Bulk, respectively.

twofold: it uses compact signatures instead of an enumeration of addresses as commit packets, and signatures are more suitable for RLE compression than address enumerations due to frequent long sequences of zeros. Figure 14 shows the commit bandwidth of Bulk normalized to that of Lazy. We see that, on average, Bulk achieves a 83% reduction in commit bandwidth.

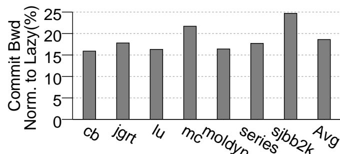

Figure 14. Commit bandwidth of Bulk normalized to the commit bandwidth of Lazy.

For TLS, we obtain qualitatively similar conclusions. We do not show data due to space limitations.

7.5. Signature Size vs Accuracy Trade-off

Finally, we evaluate the accuracy of signatures to represent sets of addresses. We choose a few signature configurations to illustrate the overall size vs accuracy trade-off. Table 8 lists the signatures we tested. For each signature, the table shows the ID, the full and average compressed size in bits, and the format. The signature in bold (*S14*) is the one we used in all previous experiments.

To assess the accuracy of a signature, we run the TM applications using that signature. We sample every bulk address disambiguation event that we know should not detect a dependence if there were no aliasing. Then, we record whether a dependence was found (false positive) or not. Figure 15 shows the fraction of false positives that such samples produced.

In the figure, each bar corresponds to one signature configuration, where the signatures are generated without any initial bit permutation on the original addresses. We see that the frequency of false positives can be high, but that it quickly decreases as the signature size increases. Within a given signature size, different configurations have different accuracies, especially for small sizes.

We then repeat the experiments with a variety of bit permutations on the original addresses before generating the signatures, as shown in Figure 2. The resulting fraction of false positives observed is shown in Figure 15 as error segments. The lower tick in an error segment corresponds to the best permutation that we tried, while the upper tick corresponds to the worst permutation. Good permutations group together bits that vary more, and map them to a large C_i bit-field. From the figure, we see that the permutation has a significant impact. Many times, it is possible to obtain better accuracy with a smaller signature and a better permutation.

ID	Full Size (Bits)	Compressed Size (Avg, in bits)	Description (See Caption)
S1	512	254	7, 7, 7, 7
S2	512	282	8, 7, 6, 5, 5
S3	512	193	5, 5, 6, 7, 8
S4	1024	290	8, 8, 8, 8
S5	1024	318	9, 8, 7, 7
S6	800	234	5, 8, 8, 8
S7	800	266	8, 5, 8, 8
S8	800	281	8, 8, 5, 8
S9	576	234	5, 8, 8, 5
S10	1344	334	9, 9, 8, 6
S11	1824	356	9, 10, 8, 5
S12	1600	353	10, 9, 6
S13	1664	353	10, 9, 7
S14	**2048**	**363**	**10, 10**
S15	2048	353	10, 9, 9
S16	2336	396	10, 10, 7, 5
S17	3072	380	10, 10, 10
S18	4096	438	11, 10, 10
S19	4096	469	11, 11
S20	4096	381	12
S21	4112	497	11, 11, 4
S22	5120	497	11, 11, 10
S23	16448	1219	13, 13, 6

Table 8. Signatures tested. The Description column shows the sizes of the bit chunks used in each of the $C_1C_2...C_n$ bit-fields of the (already permuted) address (Figure 2). These chunks are all consecutive and start from the least significant bit. The V_i bit-fields are obtained by decoding the corresponding C_i bit-fields.

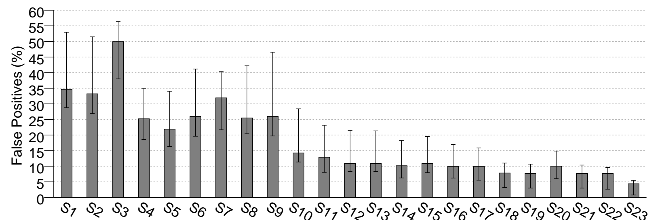

Figure 15. Fraction of false positives in bulk address disambiguations known to have no dependences. Each bar corresponds to one signature configuration, while the error segment corresponds to using different bit permutations in the address before generating the signature.

8. Related Work

There is a large volume of previous work in TLS and TM. The first hardware support for disambiguation in TLS was the Address Resolution Buffer (ARB) [7], which provided a shared table for tracking all speculative loads and stores. After that, multiple proposals have been made to move speculative data into each core's private cache or write buffer, and leverage the cache coherence protocol for disambiguation. This includes the Speculative Versioning Cache [10], the Hydra design [11], the design of Steffan and Mowry [25], and the Memory Disambiguation Table [15] among

several others. Several designs have been proposed to implement scalable conflict detection and version management for TLS [6, 24].

Herlihy and Moss [13] proposed an early architecture for TM. They used a small, fully-associative cache to buffer all speculatively-referenced data and a snoopy coherence protocol. Recently, there have been several designs for TM such as TCC [12], UTM [2], VTM [20], and LogTM [18]. They use a variety of techniques similar to TLS that hinge around leveraging the coherence protocol [2, 20, 18] and adding small buffers to track accesses [2].

Bulk differs from all of this prior work by using a signature as a compact representation of a speculative thread's access history, and by using bulk operations on signatures to perform disambiguation and speculative state management. We have argued that Bulk significantly simplifies the several mechanisms needed to enforce the data dependences across speculative threads.

Bulk uses lazy conflict detection, like TCC [12] and some TLS designs [25]. However, unlike TCC, Bulk assumes that some code will not execute in a transaction and, therefore, Bulk is compatible with a plain invalidation-based cache coherence protocol. One of the TLS designs in [25] communicated and disambiguated at the end of a task's execution, whereas Bulk allows for eager communication between tasks even though disambiguation is performed lazily. This enables higher performance.

Signatures are very similar to Bloom filters [3]. Bloom filters are employed in VTM [20] to reduce accesses to its overflow area. Specifically, VTM uses the Transaction Address Data Table (XADT) to log all speculative reads and writes. The XADT Filter (XF) is a Bloom filter that eliminates some searches of the XADT and is employed only for performance. Bulk, instead, uses signatures as the sole record of memory references.

9. Conclusions

This paper presented the concept and design of Bulk. Bulk is a novel approach to enforcing data dependences across threads in an environment with multiple, cooperating speculative threads such as TM and TLS. The cornerstone of Bulk is the use of signatures to efficiently encode a thread's access information, and signature operations in hardware that efficiently process sets of addresses. Bulk operations are inexact yet correct. They provide substantial conceptual and implementation simplicity to key mechanisms.

Compared to the state-of-the-art, some of the simplifications provided by Bulk include sending only a write signature at a commit, performing full-address disambiguation of threads in a single operation, recording speculatively-accessed addresses inexpensively with signatures, representing versions concisely without version IDs, supporting fine-grain (per word) address disambiguation with no extra storage, and committing by clearing a signature.

We evaluated Bulk in the context of TLS using SPECint2000 codes and TM using multithreaded Java workloads. We showed that, despite its simplicity, Bulk has a performance that is competitive with more complex schemes. False positives have a negligible impact on both performance and bandwidth consumption. Finally, we showed that signature configuration is a key design parameter.

Acknowledgements

We thank the anonymous reviewers and the members of the I-ACOMA group at the University of Illinois for their invaluable comments. Special thanks goes to Wonsun Ahn, Karin Strauss, Paul Sack and Brian Greskamp for their feedback on the paper.

References

[1] B. Alpern, S. Augart, S. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. McKinley, M. Mergen, J. Moss, T. Ngo, V. Sarkar, and M. Trapp, "The Jikes Research Virtual Machine Project: Building an Open-Source Research Community," *IBM Systems Journal*, November 2005.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *International Symposium on High Performance Computer Architecture*, February 2005.

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

[4] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun, "Transactional Execution of Java Programs," in *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

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

[6] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," in *International Symposium on Computer Architecture*, June 2000.

[7] M. Franklin and G. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," in *IEEE Transactions on Computers*, May 1996.

[8] M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J.-A. Gregorio, and M. Valero, "Evaluating Kilo-instruction Multiprocessors," in *Workshop on Memory Performance Issues*, June 2004.

[9] M. J. Garzaran, M. Prvulovic, J. M. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors," *ACM Transactions on Architecture and Code Optimization*, September 2006.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. Sohi, "Speculative Versioning Cache," in *International Symposium on High Performance Computer Architecture*, February 1998.

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

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

[13] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *International Symposium on Computer Architecture*, May 1993.

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

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

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "POSH: A TLS Compiler that Exploits Program Structure," in *Symposium on Principles and Practice of Parallel Programming*, March 2006.

[17] P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," in *International Conference on Supercomputing*, June 1999.

[18] 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.

[19] E. Moss and T. Hosking, "Nested Transactional Memory: Model and Preliminary Architecture Sketches," in *Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, October 2005.

[20] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *International Symposium on Computer Architecture*, June 2005.

[21] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos, "SESC Simulator," January 2005. http://sesc.sourceforge.net.

[22] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," in *International Conference on Supercomputing*, June 2005.

[23] G. Sohi, S. Breach, and T. Vijayakumar, "Multiscalar Processors," in *International Symposium on Computer Architecture*, June 1995.

[24] J. G. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A Scalable Approach to Thread-Level Speculation," in *International Symposium on Computer Architecture*, June 2000.

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

[26] M. Tremblay, "MAJC: Microprocessor Architecture for Java Computing." Hot Chips, August 1999.

[27] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew, "The Superthreaded Processor Architecture," *IEEE Transactions on Computers*, September 1999.