BULK OPERATION AND DATA COLORING FOR MULTIPROCESSOR PROGRAMMABILITY

BY

LUIS H. CEZE

B.Eng., University of Sao Paulo, 2001
M.Eng., University of Sao Paulo, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

With ubiquitous multi-core architectures, a major challenge is how to effectively use these machines. Writing parallel programs is usually very complex and error-prone. Hence, improving the programmability of parallel computer systems is a massive problem for much of our field. We believe that architecture support plays a key role in making parallel programming accessible to the masses. This thesis focuses on simple and flexible mechanisms that are used to improve the programmability of shared-memory multiprocessors.

This thesis makes two main contributions. The first contribution is efficient coarse-grain operation of multiprocessors. This allows groups of dynamic instructions to behave as a unit in a multiprocessor system. The main idea in making coarse-grain operations efficient is to hash-encode a thread's access information in a concise signature, and then support signature operations that efficiently process sets of addresses — in bulk. These operations are inexact but correct, and provide substantial conceptual and implementation simplicity. We show how to use these operations to simplify support for features that improve programmability. Specifically, we discuss designs for Transactional Memory, Thread-Level Speculation and high-performance sequential consistency. We also suggest other possible uses of bulk operations to illustrate its versatility.

The second contribution of this thesis is architecture support and programming model for a data-centric approach to thread synchronization. In *Data-Centric* Synchronization (DCS), the programmer uses local reasoning to assign synchronization constraints to data. Based on these, the system automatically infers critical sections and inserts synchronization operations.

To "pequena" Karin, my source of inspiration, balance and love. She is so important to me that anything would be an understatement.

Acknowledgments

I would like to thank my advisor, Prof. Josep Torrellas, for being such a great mentor. Also, Prof. Torrellas did more than mentoring me, he helped me tremendously in debugging my ideas and improving the quality of my text. I learned invaluable skills working for him. He taught me how to focus on the most important things! He also showed me the importance of being meticulous and tenacious when it comes to writing (and hopefully publishing) papers.

I would also like to thank my thesis committee members, Prof. Marc Snir, Prof. Sarita Adve, Prof. David Padua, Prof. Craig Zilles, Prof. Wen-Mei Hwu and Dr. Calin Cascaval for taking the time to give me feedback and being so flexible when scheduling my final exam. Prof. Marc Snir was extremely helpful in guiding me during the revision process of this thesis.

Six years ago I was starting my first internship at IBM Research. During that time I learned a lot about doing research and most importantly, met several people that encouraged me to get a PhD and mentored me through graduate school. That first opportunity was given to me by Dr. Jose E. Moreira, who became a great mentor and friend. I also learned a lot from Dr. Călin Caşcaval, Dr. José Castaños e Dr. Gheorghe Almasi. I still remember Călin telling me "read the manual!", José Castaños coaching me for the GRE and Gheorghe suggesting improvements to my code. I went back to IBM several other summers and during the Summer of 2004 the idea of Bulk signatures was conceived (in Călin's office). That idea became a centerpiece of this thesis. My constant interaction with Călin's and close collaboration during my time at school was extremely valuable to me. I was also very fortunate to collaborate with Dr. Christoph von Praun, whose expertise on synchronization

was invaluable in developing the work on coloring for data-centric synchronization. Here I express my deepest gratitude to IBM Research, for giving me numerous opportunities during my internships and for supporting me financially for two years with a PhD Fellowship. But most especially, I would like to thank IBM Research for being the place where I met many people that mentored me and became so important to me during my graduate student career.

I had many co-workers during my time in graduate school. Without their help, I would probably have taken much longer to get the work on this thesis done, if at all. James Tuck was an incredible source of help on the work on Bulk signatures, we had many insightful discussions and very productive times. I had a thrill collaborating with him. Pablo Montesinos, the tallest student in computer architecture at this time, was very helpful with the work on data-centric synchronization and also on the *BulkSC* work. Jose Renau, now a professor at UC Santa-Cruz taught me a lot about thread-level speculation when we worked so hard on our TLS system. Also, his work on the SESC simulator was instrumental in evaluating the work on this thesis. Other people with whom I did not work directly with but benefited a lot from their feedback are: Paul Sack, Radu Teodorescu, Brian Greskamp, Abhishek Tiwari, Smruti Sarangi and Wonsun Ahn. Spending countless hours in the I-ACOMA office was a lot of fun, I will never forget it. Paul introduced me to the pleasures of prime coffee, roasted and ground fresh. The taste was great, but the caffeine was invaluable! Smruti enlightened me about Indian culture and his knowledge of international politics. Brian, the nicest guy in the office, was always a pleasure to brainstorm with and he always listened patiently to my ramblings. I-ACOMers, you are great friends and a great source of strength, laughter and ears to listen to my complaints!

I also owe my success in this doctorate program to Prof. Wilson Ruggiero and Prof. Tereza Carvalho from the University of São Paulo. They believed in me and encouraged me to always do better, besides giving me many opportunities during my time in their lab.

Sheila Clark made my life at UIUC much easier by being so efficient with all the administrative matters. Also, I am very thankful to her for helping me with my job search process

in many ways, but especially by assisting the faculty members that were writing letters of recommendation on my behalf.

We are social beings. Well, graduate students are almost social beings. Nevertheless, friends are the most important part of life. I made many friends at UIUC with whom I shared jokes, beer, trips, meals, movies and other forms of time wasting. Paul Sack was always up for some beef! He was also the source of the best and worse jokes I have ever heard. I had many joyful happy hours with Pierre Salverda and Naveen Neelekantam when we talked about computer architecture, good food and whined about paper rejections.

My family always valued education. More than just giving me the best education they could find, they taught me how valuable it is. My father, also an engineer, always encouraged me to build things and solve problems, he also introduced me to computers and how fun they can be. He started it all. I thank my family for being so loving and so present in my life even from a distance. Living far from home is hard, but luckily technology is making it much less painful with affordable videoconferencing — thanks to all the people involved in H.264! I am always looking forward to our next videoconference. It allows me to have much closer interaction with my family in Brazil. However, it did allow my mother to see that my hair is starting to turn gray.

Without Karin nothing would have been possible. She is the most important person in my life. Also, she happens to be one of the smartest people I've ever met. She gave invaluable suggestions, feedback and insights that impacted the work presented in this thesis. She helped me technically, personally and emotionally, I am very grateful. Karin is the best partner in life somebody could ask for. I love her. Period.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chip-multiprocessors (CMPs) are becoming ubiquitous, as virtually all processor manufacturers offer designs with multiple cores in the same chip. There are two main reasons for this: (i) lower design complexity than larger monolithic cores and (ii) better power efficiency (MIPS per Watt). However, sequential programs are unlikely to run much faster in these architectures. Therefore, in order to harvest the vast computational resources offered by CMPs, we need to make parallel programs ubiquitous as well.

Not surprisingly, there is growing expectation that parallel programming will become popular. Unfortunately, the vast majority of current application programmers find parallel programming too difficult. There are several reasons why parallel programming is considered hard. First, it is necessary to identify sources of parallelism. Then, it is necessary to deal with thread synchronization, which is typically error-prone (deadlocks, races and live-locks). Also, relaxed memory consistency models, which are commonly used in current systems, often confuse programmers. Finally, to make matters worse, parallel programs are notoriously hard to debug due to non-deterministic behavior.

Improving the programmability of multiprocessor systems is a major problem in computer science and engineering. We believe that architecture plays a key role in attacking this problem. At the lowest level, the architecture should provide a simple machine model, which we believe is a shared-memory paradigm with a simple, intuitive memory consistency model such as sequential consistency [45, 20]. On top of that simple model, the architecture can provide primitives for better programming models. This is especially true for thread synchronization, with features like transactional memory [37]. The architecture can also pro-

vide mechanisms that decrease the correctness requirements imposed on the programmer. For example, if the architecture supports thread-level speculation, the hardware provides a safety net that lets the programmer optimistically parallelize a program without worrying about occasional dependences that could potentially violate correctness. Finally, the hardware can offer hooks to aid in software debugging, making it easier to find bugs in large parallel programs.

One important aspect of architecture support for programmability is that it is not transparent to the software. Improving programmability requires rethinking the hardware/software interface, making it a joint effort between research in architecture, programming models, compilers and tools.

In this thesis, we propose a set of architectural primitives that aim to improve the programmability of shared-memory multiprocessors. When devising these new features, we paid special attention to keeping complexity low — one of the arguments in favor of CMPs.

At a high level, this thesis makes two major contributions. The first contribution is efficient coarse-grain operation of multiprocessors. This allows groups of dynamic instructions to behave as a unit in a multiprocessor system. This is desirable because, as we will show, it simplifies hardware design of many programmability-enhancing mechanisms. We make coarse-grain operation efficient by encoding sets of addresses in a Bloom-filter-based [9] signature and by defining operations with these signatures, such as union or intersection. These operations are conceptually simple and very easy to implement in hardware. We show how to use these operations to support transactional memory, thread-level speculation and high-performance sequential consistency. We also suggest other possible uses of bulk operations to illustrate its versatility.

The second major contribution of this thesis is architecture support and a programming model for a data-centric approach to thread synchronization. In data-centric synchronization, the programmer assigns synchronization constraints to program data, as opposed to

code, like what is done typically with locks or transactions. Based on these, the system automatically infers critical sections, freeing the programmer from having to annotate all places in the code where shared data might be accessed.

This thesis is organized in two parts, Part I describes bulk operation for multiprocessors and its application to transactional memory, thread-level speculation and high-performance sequential consistency. The text in Part I is mostly based on [15] and [16]. Part II describes architectural support for data-centric synchronization, and it also outlines ideas on how to use the same support for debugging parallel programs. The text in Part II was mostly based on [13].

Part I

Bulk Operation for Multiprocessors

Chapter 2

Motivation for Bulk Operations

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., [5, 34, 37, 54, 60]), the speculative threads are obtained from parallel programs, and the emphasis is typically on easing programmability. In TLS (e.g., [33, 44, 49, 63, 64, 67, 69, 70, 74]), the speculative threads are extracted from a sequential program, and the goal is to speed-up the program. Finally, Checkpointed multiprocessors [14, 25, 42] 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., [26]), 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.

Our goal in this context is to simplify the conceptual and implementation complexity of these mechanisms. For that, we employ a Bloom-filter-based [9] 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.

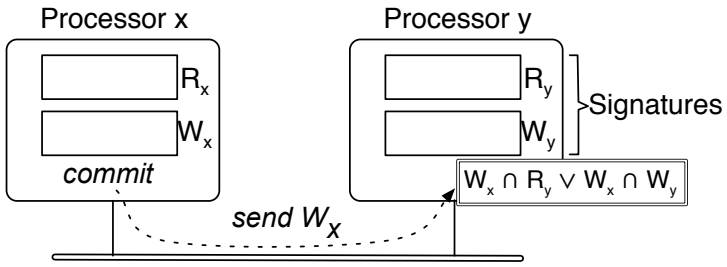


Figure 2.1: Example of an operation in Bulk.

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 individual addresses. As an example, Figure 2.1 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 Part I of this thesis, we describe three 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. Third, we propose and evaluate BulkSC, which uses bulk operations to enforce sequential consistency at a coarse grain in multiprocessors.

Part I is organized as follows: Section 2.1 presents a brief background on speculative multithreading and motivates bulk operations; Chapter 3 presents signatures and basic operations on them; Chapter 4 details the Bulk architecture; Chapter 5 discusses using Bulk for TM and TLS, including a detailed evaluation; Chapter 6 describes and provides an evaluation of the BulkSC memory ordering framework.

2.1 Speculative Threads in Multiprocessors

Both TLS and TM are environments with multiple speculative threads. In TLS (e.g. [33, 44, 49, 63, 64, 67, 69, 70]), 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., [5, 34, 37, 54, 60]), 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 earlier, 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.

Chapter 3

Address Signatures and Bulk Operations

To reduce the implementation complexity of the operations described in Section 2.1, this thesis 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 [9].

In more formal terms, we want to represent n-elements sets of addresses from a very large universe of addresses U, where $|U| = u >> n$. A signature is a bit vector S of m bits, where $m << u$. We define a hash function H that encodes an addresses $a \in U$ in a signature, H is such that $H(a) \in \{0,1\}^m$ with k bits set. If A is an address set, $H(A) = \bigvee_{a \in A} H(a)$. We want to choose H so as to minimize the likelihood that two randomly chosen distinct sets from U map to the same signature. We define the inverse of H, H^{-1},

such that $H^{-1}(S) = \{a : H(a) \wedge S = H(a)\}$, this way, $A \subseteq H^{-1}(H(A))$. Properties of H are such that if $A \subset B$, $H(A) \wedge H(B) = H(A)$ and $H(A \cup B) = H(A) \vee H(B)$. Hence, we call a signature S a superset encoding of a set of addresses from U. We call a signature S empty if $H^{-1}(S) = \emptyset$.

Figure 3.1 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 $C_1, \ldots, C_k$. 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.

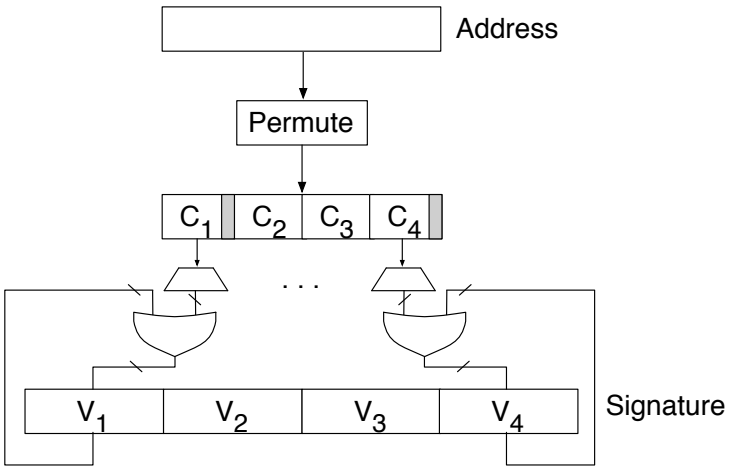


Figure 3.1: Adding an address to a signature.

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 3.1. 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 ad-

dresses 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 Bandwidth optimizations
$\in$	Membership of an address in a signature	Address disambiguation with individual address
δ	Signature decoding into sets (exact)	Signature expansion

Table 3.1: 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 adding a to 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 precisely 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 if we select 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 3.1 also lists a sample use of each operation. We will discuss the uses in Chapter 4. Finally, Figure 3.2 shows how these operations are implemented.

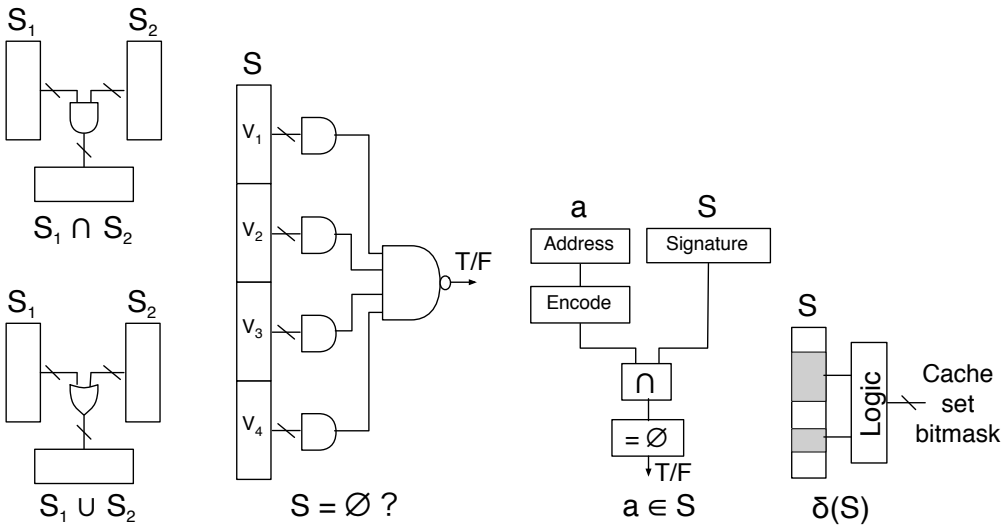


Figure 3.2: Implementation of the primitive bulk operations on signatures.

3.3 Signature Expansion

There is one other important Bulk operation that is composed of two of the primitive operations in Table 3.1. 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 3.3 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.

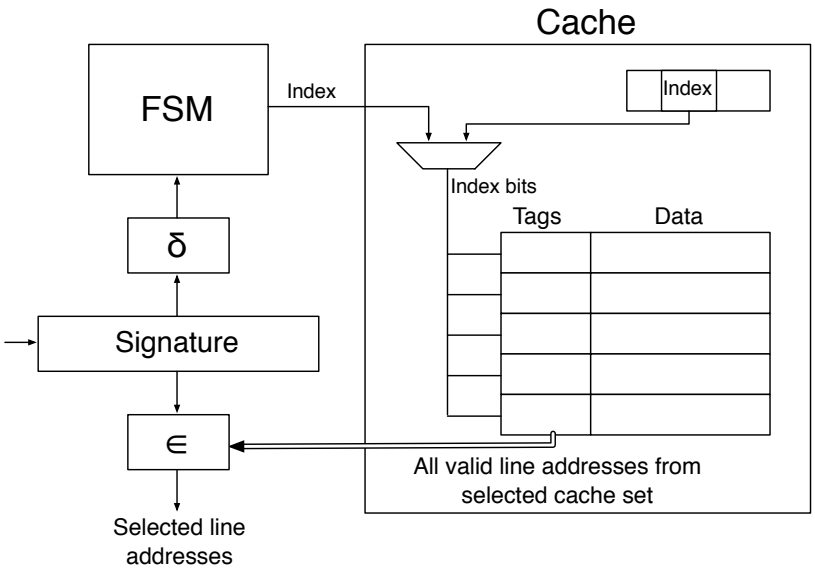


Figure 3.3: Implementation of signature expansion.

Chapter 4

An Architecture for Bulk Disambiguation

Based on the primitive described in Chapter 3, we can now build the complete *Bulk* architecture. Bulk presumes a multiprocessor with an invalidation-based cache coherence protocol. For generality, an application can run both non-speculative and speculative threads. The former send 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 (see Section 2.1) as described in Section 2.1.

In Bulk, every speculative thread has a *Read* (R) and a *Write* (W) signature in hardware (Figure 2.1). At every load or store, the hardware adds the requested address to R or W, respectively, as shown in Figure 3.1. 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 or interacting with an

arbiter). When the thread knows that its commit will proceed, it sends out its write signature W_C so that it can be disambiguated against all other threads in the system (Figure 2.1) and it sets its W_C and R_C signatures to empty. This is shown in Figure 4.1(a).

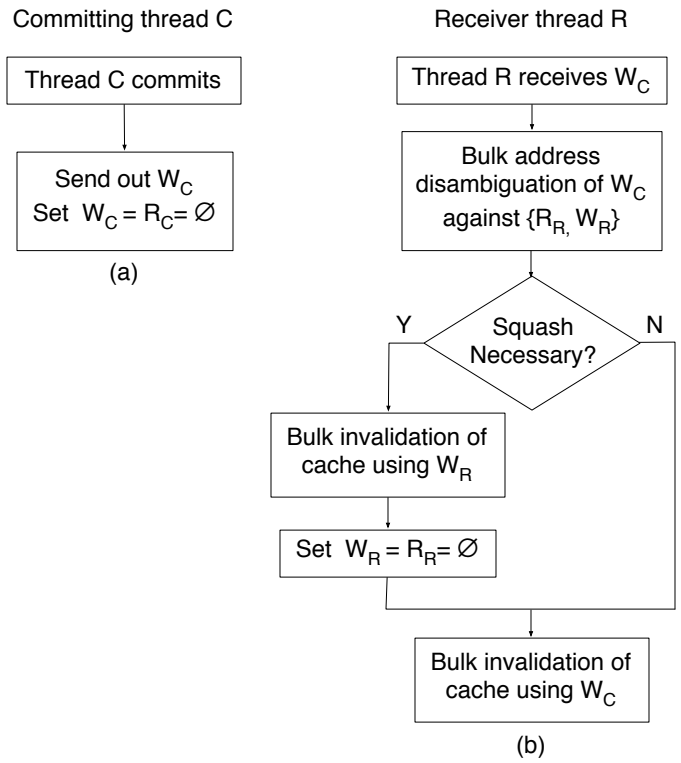


Figure 4.1: Flowchart of the commit process: committing thread (a) and receiver thread (b).

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 4.1(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 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

$$W_C \cap R_R \neq \emptyset \quad \vee \quad W_C \cap W_R \neq \emptyset \tag{4.1}$$

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 of potential imprecision issues when disambiguating at a fine grain (discussed in detail 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 5.3 and Section 6.6, 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.

[1]In TLS, other cache lines may be invalidated as well (Section 5.2).

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 cache 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 [19], 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 4.2). 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 4.2), and the resulting line is written to the cache. Note that this process requires no cache modifications — not even per-word access bits.

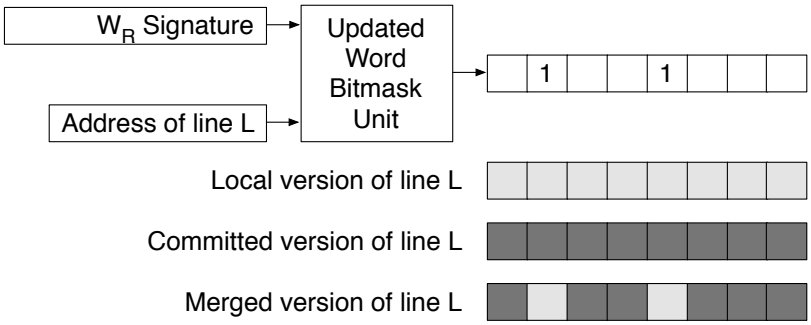


Figure 4.2: Merging lines partially updated by two speculative threads.

From this discussion, it can be deduced why the $W_C \cap W_R \neq \emptyset$ component of Equation 4.1 is required even in word-level disambiguation. Specifically, W_R is conservative due to aliasing, and the word bitmask of Figure 4.2 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 4.3 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.

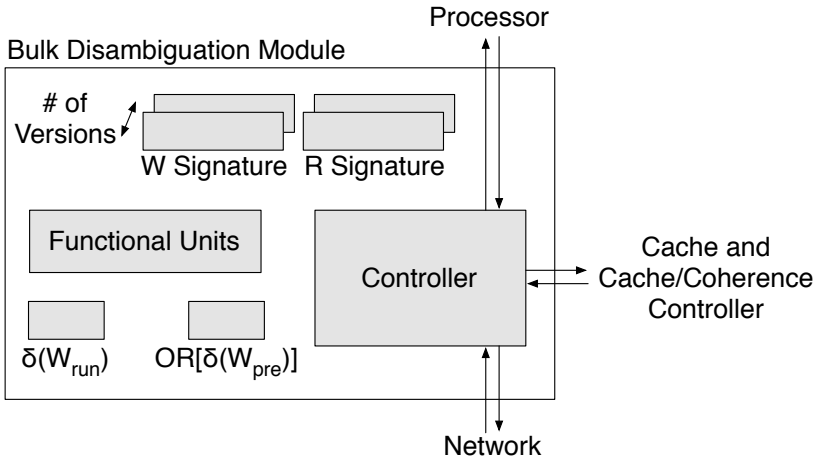


Figure 4.3: Overview of the Bulk Disambiguation Module (BDM).

The BDM also has a set of functional units. They perform the primitive bulk operations of Table 3.1, the Signature Expansion of Section 3.3, and the bitmask of updated words of Section 4.4.

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, $W_1 \cap W_2 = \emptyset$.

4.5.1 Write-through and Shared Caches

The Bulk architecture has assumed a writeback L1 cache, where speculative dirty data can be conveniently held. To make the Bulk architecture work properly with write-through caches, it is necessary to provide some mechanism to store speculative dirty data. This could be done in a variety of ways, for example, provide custom storage between the L1 ad L2 to hold speculative data. Another alternative could be to have the L2 cache hold speculative data.

The Bulk architecture has also assumed that the L1 cache is not shared by multiple cores or by multiple contexts of execution in a core with support for simultaneous multithreading. However, such an environment can be supported relatively easily. We envision using the mechanisms for holding multiple speculative versions in the same cache as described in Section 4.5. One very important requirement of supporting shared caches is to keep speculative dirty data visible only to its owner context. One way of accomplishing this is to have the BDM block accesses to a speculative dirty line that belongs to another context much in the same way it blocks external accesses. This could be done by checking if the read address is present in the write signature of the other contexts that share the same cache. Finally, holding speculative data in L2 increase the importance of supporting shared caches, since the L2 is more likely to be shared by multiple contexts.

4.6 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 5.3.5.

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 5.3.5.

4.6.1 Dynamic Adjustment of Signature Parameters

Memory access patterns may change from application to application and between phases in the same application. These patterns may affect how accurate read/write signatures are and potentially cause too many unnecessary restarts. If an application has a pathological signature behavior, dynamic adjustment of the signature parameters might be a solution. If the signature encoding mechanism is made configurable it could be changed to better accommodate the access patterns of the running application and therefore reduce unnecessary squashes.

If signatures generated with different parameters can still be compared, the transition in signature parameters is less costly, because the system does not have to be stopped to make sure all signatures in flight in the system are of the same configuration. To enable that using our signature format, it is necessary to follow some guidelines on the shape of the permutation and the address decoding mechanism.

Varying the size of signatures also has extra advantages. If signatures can be made smaller for a given application while keeping the accuracy similar, that would have bandwidth advantages. Also, if the way the signature file is implemented in hardware allows having more signatures if they are smaller, that would definitely benefit applications that tend to have small but deeply nested transactions or tasks. Having smaller signatures would allow a fixed signature file to hold more signatures.

4.7 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.

4.7.1 Compact Representation of Address Sets

Bulk represents the set of addresses accessed speculatively very concisely. This simplifies the hardware implementation of several operations. Table 4.1 lists some of the key simplifications in Bulk. We discuss each one in turn.

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 4.1: 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 5.1.2).

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 4.3). 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.

4.7.2 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 3.1, 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.

Chapter 5

Using Bulk for TM and TLS

This chapter examines implementation details relevant to using Bulk for TM and TLS.

5.1 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.

5.1.1 Transaction Nesting

Figure 5.1 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 [55], 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.

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

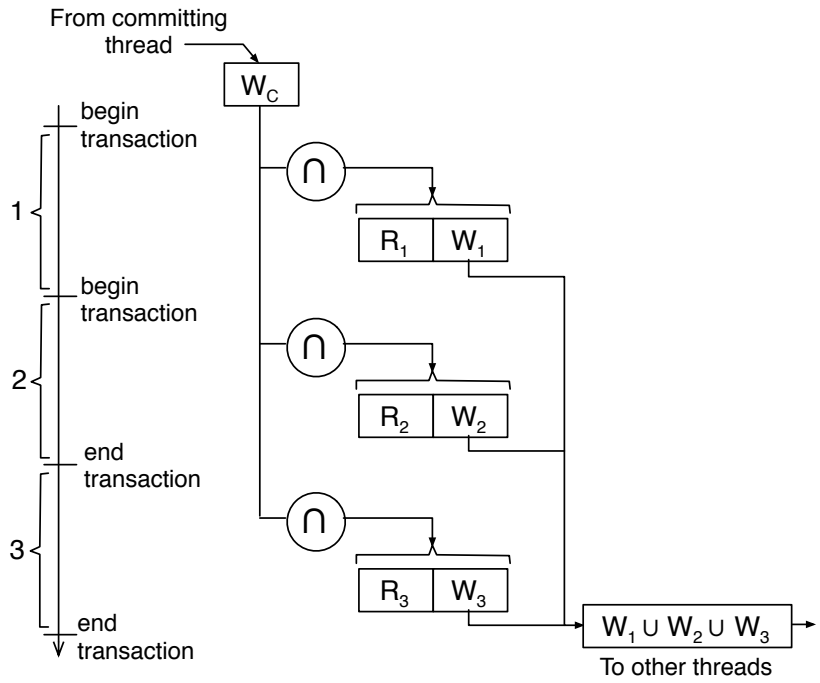


Figure 5.1: Supporting nested transactions in Bulk.

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.

Note that while the example in Figure 5.1 shows an implementation with three pairs of signatures, an implementation with only 2 is possible, but could have lower performance. For example, section *3*'s references could be encoded in R_1 and W_1. However, if a conflict happened with a reference in section *3*, all sections of the example would be squashed.

5.1.2 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 [5, 60]. In a context switch, many conventional schemes move the cached state of the preempted speculative thread to the overflow area [5, 60].

Bulk reduces the complexity and the performance overhead of having to deal with overflows and context switches. The three reasons are shown in Table 5.1. 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 5.1: 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 4.3), 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., [60]) — 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.

5.2 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 5.2). From the point of the spawn on, both signatures are updated at every write.

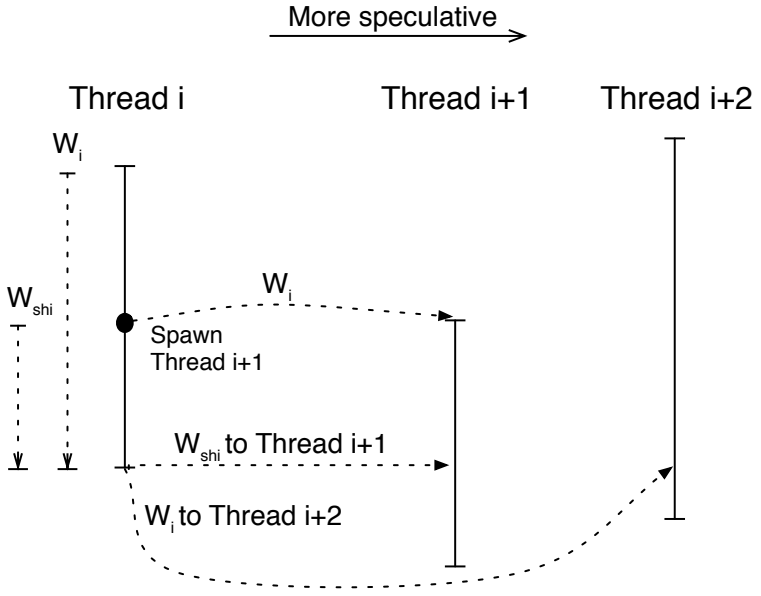


Figure 5.2: Supporting Partial Overlap in TLS.

Secondly, when a thread commits, it sends both its write signature W and its shadow one W_{sh} (Figure 5.2). 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 5.2). In the receiving processor, the spawn operation includes a bulk invalidation of the clean cached lines whose addresses are in W.

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.

Alternatively, this same feature could be implemented by freezing the collection of W when a thread spawns its first child and start collecting W_{sh}. When the thread commits, its first child uses W_{sh} for disambiguation and all other threads use $W \cup W_{sh}$.

5.3 Evaluation

In this section, we evaluate Bulk in the context of both TM and TLS. After presenting our evaluation setup in Section 5.3.1, we show that Bulk induces a very small performance degradation in Section 5.3.2. We characterize the main aspects of the bulk operations in Section 5.3.3. We then present bandwidth issues in Section 5.3.4. Finally, we show the trade-offs of signature encoding in Section 5.3.5.

5.3.1 Evaluation Setup

For the TLS evaluation, we compile the applications using a fully automatic profile-based TLS compilation infrastructure [47]. We run the binaries on an execution-driven simulator [62] with detailed processor core and memory system models, including all TLS operation overheads, such as thread spawn, thread squash, and versioning support. The machine simulated has a 32-bit address space (same as the TM experiments). 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 [4] to add *begin* and *end* transaction annotations to Java programs. We convert lock-based constructs into transactions using methods similar to [12]. 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 [5, 54, 60]. The TM simulation includes a detailed memory model. As listed in Table 5.2, 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 5.2: Java applications used in the TM experiments.

Table 5.3 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 5.2) and for overflows and context switches in TM (Section 5.1.2). It does not include support for partial rollback of nested transactions (Section 5.1.1).

5.3.2 Performance

Figure 5.3 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

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 5.6 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 5.3: Architectural parameters used in the TM/TLS 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.

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 5.3 also includes the performance of Bulk without the support for Partial Overlap discussed in Section 5.2. 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 5.4 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

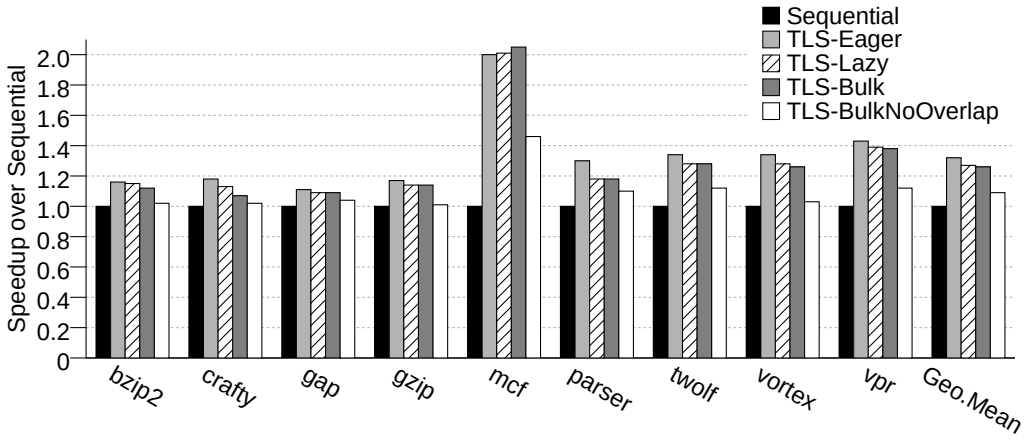


Figure 5.3: Performance of Eager, Lazy, and Bulk in TLS.

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 5.5(a), where two threads read and write the same location inside a transaction, and they keep squashing each other repeatedly[1]. The second reason involves a situation where a squash happens in Eager but not in Lazy, as shown in Figure 5.5(b).

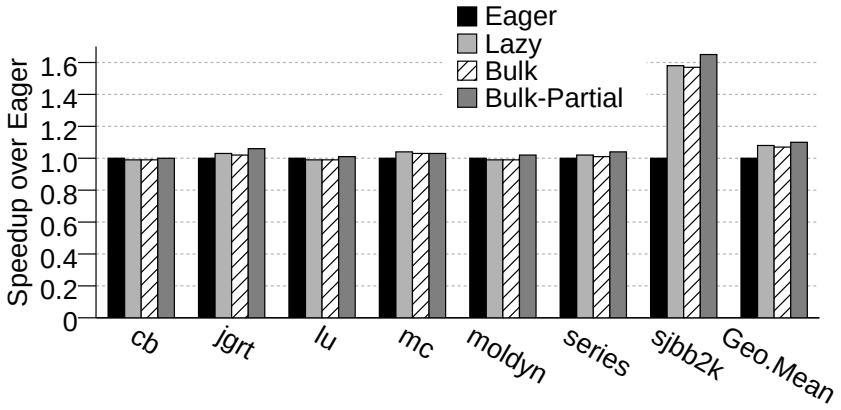
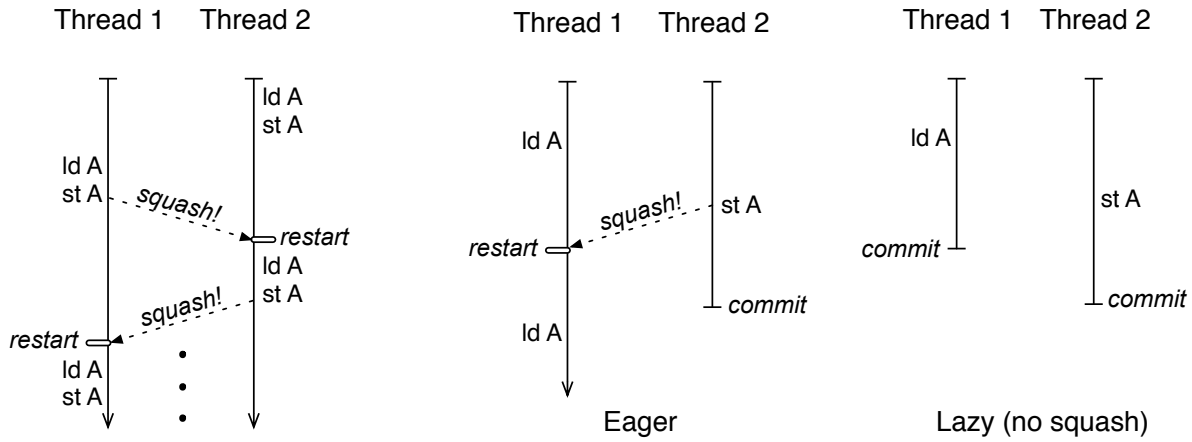


Figure 5.4: Performance of the different schemes in TM.

The Bulk-Partial bar in Figure 5.4 shows the performance of supporting partial rollback in nested transactions (Section 5.1.1). The plot shows that the impact of partial rollback

[1]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.



(a) No forward progress in Eager.

(b) Squash happens in Eager but not in Lazy.

Figure 5.5: Examples of code patterns from SPECjbb2000 where the performance of Eager suffers.

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.

5.3.3 Characterization of Bulk

Table 5.4 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.

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

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 5.4: Characterization of Bulk in TLS. The averages are over all dynamic tasks.

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 5.3.

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 5.5 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 5.5: Characterization of Bulk in TM.

The *False Positives* columns show information similar to the corresponding columns of Table 5.4. 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 5.1.2). We can see that Bulk is very effective at avoiding accesses to the overflow area.

5.3.4 Bandwidth Usage in TM

We study the bandwidth usage in TM by looking at both the total bandwidth and the commit bandwidth usage. Figure 5.6 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*).

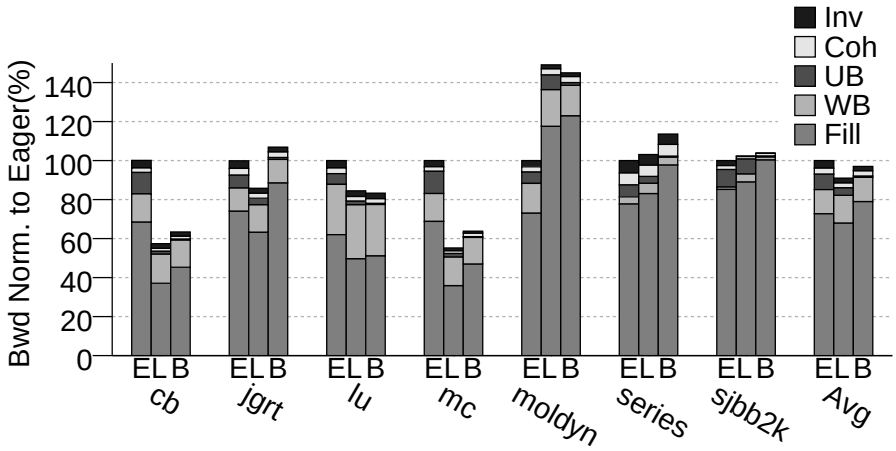


Figure 5.6: Bandwidth usage breakdown in TM. Underneath the bars, *E*, *L* and *B* refer to Eager, Lazy and Bulk, respectively.

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 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 5.7 shows the commit bandwidth of Bulk normalized to that of Lazy. We see that, on average, Bulk achieves a 83% reduction in commit bandwidth.

For TLS, we obtain qualitatively similar conclusions.

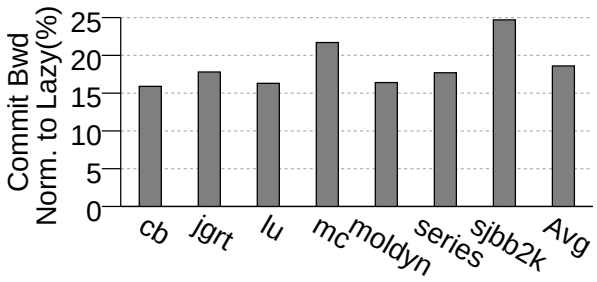


Figure 5.7: Commit bandwidth of Bulk normalized to the commit bandwidth of Lazy.

5.3.5 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 5.6 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 5.8 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 3.1. The resulting fraction of false positives observed is shown in Figure 5.8 as error segments. The lower tick in an error segment corresponds to the best permutation that we tried, while the upper tick corresponds

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 5.6: 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 3.1). 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.

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.

5.4 Related Work on TM and TLS

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) [24], which provided a shared table for tracking all speculative loads and stores. After that, multiple proposals

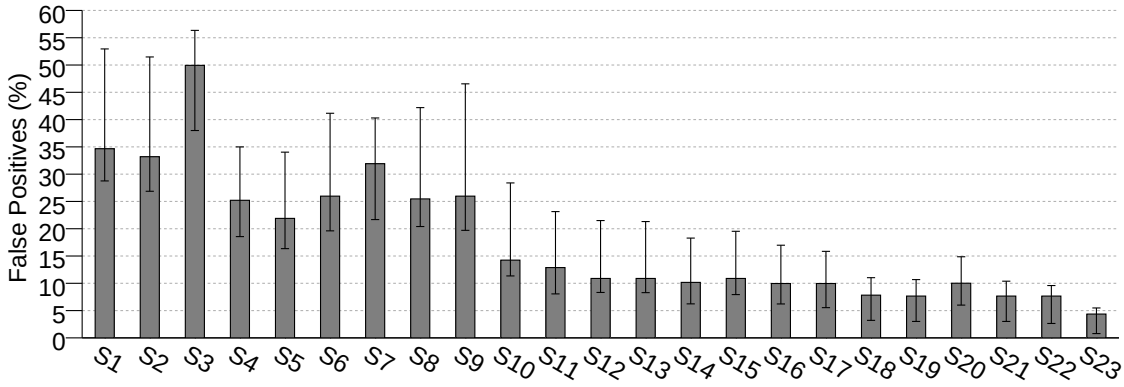


Figure 5.8: Fraction of false positives in bulk address disambiguations known to have no dependences. Each bar corresponds to one signature configuration, while the error segment corresponds to using different bit permutations in the address before generating the signature.

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 [31], the Hydra design [33], the design of Steffan and Mowry [66], and the Memory Disambiguation Table [44] among several others. Several designs have been proposed to implement scalable conflict detection and version management for TLS [19, 67].

Herlihy and Moss [37] 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 [34], UTM [5], VTM [60], and LogTM [54]. They use a variety of techniques similar to TLS that hinge around leveraging the coherence protocol [5, 60, 54] and adding small buffers to track accesses [5].

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 [34] and some TLS designs [66]. 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 [66] 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 [9]. Bloom filters are employed in VTM [60] 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.

Chapter 6

BulkSC: Bulk Enforcement of Sequential Consistency

It is widely accepted that the most intuitive memory consistency model, and the one that most programmers assume is Sequential Consistency (SC) [45, 20]. SC requires that all memory operations of all processes appear to execute one at a time, and that the operations of a single process appear to execute in the order described by that process's program. Programmers prefer this model because it offers the same relatively simple memory interface as a multitasking uniprocessor [38].

Despite this advantage of SC, manufacturers such as Intel, IBM, AMD, Sun and others have chosen to support more relaxed memory consistency models [1]. Such models have been largely defined and used to facilitate implementation optimizations that enable memory access buffering, overlapping, and reordering. It is felt that a straightforward implementation of the stricter requirements imposed by SC on the outstanding accesses of a processor impairs performance too much. Moreover, it is believed that the hardware extensions that are required for a processor to provide the illusion of SC at a performance competitive with relaxed models are too expensive.

To ensure that the upcoming multiprocessor hardware is attractive to a broad community of programmers, it is urgent to find novel implementations of SC that both are simple to realize and deliver performance comparable to relaxed models. In this thesis, we propose one such novel implementation.

We call our proposal *Bulk Enforcement of SC* or *BulkSC*. The key idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. Then, the hardware enforces SC at the coarse grain of chunks rather than at

the conventional, fine grain of individual memory accesses. Enforcing SC at chunk granularity can be realized with simple Bulk signature hardware and delivers high performance. Moreover, to the program, it appears as providing SC at the memory access level.

BulkSC keeps hardware simple mainly by leveraging two sets of mechanisms: those of Bulk operations (Section 4) and those of checkpointed processors (e.g., [3, 14, 21, 42, 50, 57, 65]). Together, they largely *decouple* memory consistency enforcement from processor structures. *BulkSC* delivers high performance by allowing full memory access reordering and overlapping within chunks and across chunks.

In addition to presenting the idea and main implementation aspects of *BulkSC*, we describe a complete system architecture that supports it with a distributed directory and a generic network. Our results show that *BulkSC* delivers performance comparable to Release Consistency (RC) [28]. Moreover, it only increases the network bandwidth requirements by 5-13% on average over RC, mostly due to signature transfers and squashes.

This chapter is organized as follows. Section 6.1 gives a background on sequential consistency; Section 6.2 presents the main idea of Bulk Enforcement of SC; Sections 6.3, 6.4, and 6.5 describe the complete architecture; Section 11.3 evaluates it; and Section 6.7 discusses related work.

6.1 Background on Sequential Consistency

As defined by Lamport [45, 20], a multiprocessor supports SC if the result of any execution is the same as if the memory operations of all the processors were executed in some sequential order, and those of each individual processor appear in this sequence in the order specified by its program. This definition comprises two ordering requirements:

Req1. Per-processor program order: the memory operations from individual processors maintain program order.

Req2. Single sequential order: the memory operations from all processors maintain a single

sequential order.

SC provides the simple view of the system shown in Figure 6.1. Each processor issues memory operations in program order and the switch connects an arbitrary processor to the memory at every step, providing the single sequential order.

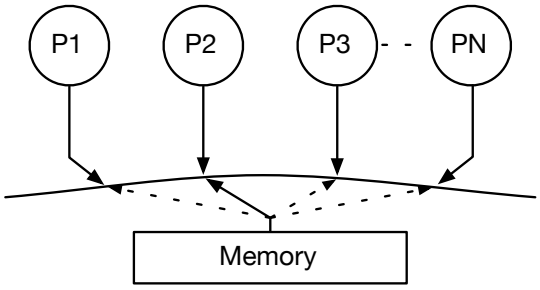


Figure 6.1: Programmer's model of SC.

A straightforward implementation of SC involves satisfying the following requirements [1]: (i) a processor must ensure that its previous memory operation is complete before proceeding with its next one in program order, (ii) writes to the same location need to be made visible in the same order to all processors, and (iii) the value of a write cannot be returned by a read until the write becomes visible to all processors — for example, when all invalidations or updates for the write are acknowledged. Since requirement (i), in particular, limits performance significantly, several optimizations have been proposed to enable memory accesses to overlap and reorder while keeping the illusion of satisfying these requirements.

Gharachorloo *et al.* [27] proposed two techniques. The first one is to automatically prefetch ownership for writes that are delayed due to requirement (i). This improves performance because when the write can be issued, it will find the data in the cache — unless the location is invalidated by another thread in between. The second technique is to speculatively issue reads that are delayed due to requirement (i) — and roll back and reissue the read (and subsequent operations) if the line read gets invalidated before the read could have been originally issued. This technique requires an associative load buffer that stores the speculatively-read addresses. Incoming coherence requests and local cache displacements

must snoop this buffer, and flag an SC violation if their address matches one in the buffer. The MIPS R10000 processor supported SC and included this technique [79]. Later, Cain *et al.* [11] proposed an alternate implementation of this technique based on re-executing loads in program order prior to retirement.

Ranganathan *et al.* [61] proposed Speculative Retirement, where loads and subsequent memory operations are allowed to speculatively retire while there is an outstanding store, although requirement (i) would force them to stall. The scheme needs a history buffer that stores information about the speculatively retired instructions. The per-entry information includes the access address, the PC, a register, and a register mapping. Stores are not allowed to get reordered with respect to each other. As in the previous scheme, the buffer is snooped on incoming coherence actions and cache displacements, and a hit is a consistency violation that triggers an undo.

Gniady *et al.* [30] proposed *SC++*, where both loads and stores can be overlapped and reordered. The ROB is extended with a similar history buffer called Speculative History Queue (SHiQ). It maintains the speculative state of outstanding accesses that, according to requirement (i), should not have been issued. To reduce the cost of checking at incoming coherence actions and cache displacements, the scheme is enhanced with an associative table containing the different lines accessed by speculative loads and stores in the SHiQ. Since the SHiQ can be very large to tolerate long latencies, in [29], they propose *SC++lite*, a version of SC++ that places the SHiQ in the memory hierarchy.

While these schemes progressively improve performance — SC++ is nearly as fast as RC — they also increase hardware complexity substantially because they require (i) associative lookups of sizable structures and/or (ii) tight coupling with key processor structures such as the load-store queue, ROB, register files, and map tables.

6.2 Bulk Enforcement of Sequential Consistency

Our goal is to support the concept of SC expressed in Figure 6.1 with an implementation that is simple and enables high performance. We claim that this is easier if, rather than conceptually turning the switch in the figure at individual memory access boundaries, we do it only at the boundaries of groups of accesses called Chunks. Next, we define an environment with chunks and outline a chunk-based implementation of SC.

6.2.1 An Environment with Chunks

Consider an environment where processors execute sets of consecutive dynamic instructions (called *Chunks*) as a unit, in a way that each chunk appears to execute atomically and in isolation. To ensure this perfect encapsulation, we enforce two rules:

Rule1. Updates from a chunk are not visible to other chunks until the chunk completes and commits.

Rule2. Loads from a chunk have to return the same value as if the chunk was executed at its commit point. Otherwise, the chunk would have "observed" a changing global memory state while executing.

In this environment, where a chunk appears to the system and is affected by the system as a single memory access, we will support SC if the following "chunk requirements" — which are taken from Section 6.1 using chunks instead of memory accesses — hold:

CReq1. Per-processor program order: Chunks from individual processors maintain program order.

CReq2. Single sequential order: Chunks from all processors maintain a single sequential order.

In this environment, some global interleavings of memory accesses from different processors are not possible (Figure 6.2). However, all the resulting possible executions are sequentially consistent at the individual memory access level.

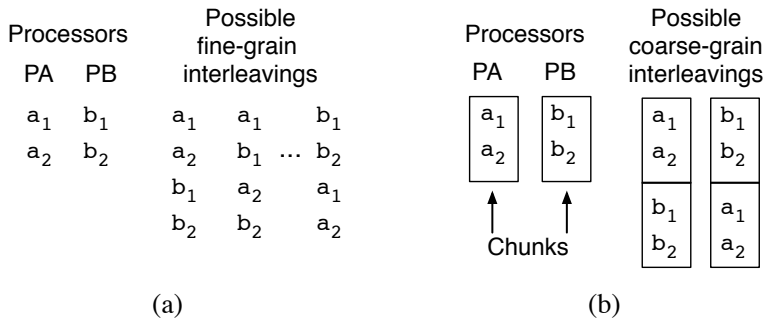


Figure 6.2: Fine (a) and coarse-grain (b) access interleaving.

6.2.2 Implementing SC with Chunks

A trivial implementation that satisfies these two SC requirements involves having an arbiter module in the machine that commands processors to execute for short periods, but only one processor at a time. During the period a processor runs, it executes a chunk with full instruction reordering and overlapping. In reality, of course, we want all processors to execute chunks concurrently. Next, we describe a possible implementation, starting with a naive design and then improving it.

Naive Design

We divide our design into two parts, namely the aspects necessary to enforce the two rules of Section 6.2.1, and those necessary to enforce the two chunk requirements of SC from Section 6.2.1.

Enforcing the Rules for Chunk Execution. To enforce *Rule1*, we use a cache hierarchy where a processor executes a chunk speculatively as in TLS or TM, buffering all its updates in its cache. These buffered speculative updates can neither be seen by loads from other chunks nor be displaced to memory. When the chunk commits, these speculative updates are made visible to the global memory system.

To enforce *Rule2*, when a chunk that is executing is notified that a location that it has accessed has been modified by a committing chunk, we squash the first chunk.

To see how violating *Rule2* breaks the chunk abstraction, consider Figure 6.3(a). In the figure, *C1* in processor *P1* includes *LD A* followed by *LD B*, while *C2* in *P2* has *ST A* followed by *ST B*. Suppose *P1* reorders the loads, loading *B* before *C2* commits. By *Rule1*, *C1* gets committed data, namely the value before *ST B*. If *C2* then commits — and therefore makes its stores visible — and later *C1* loads *A*, then *C1* would read the (committed) value of *A* generated by *C2*. We would have broken the chunk abstraction.

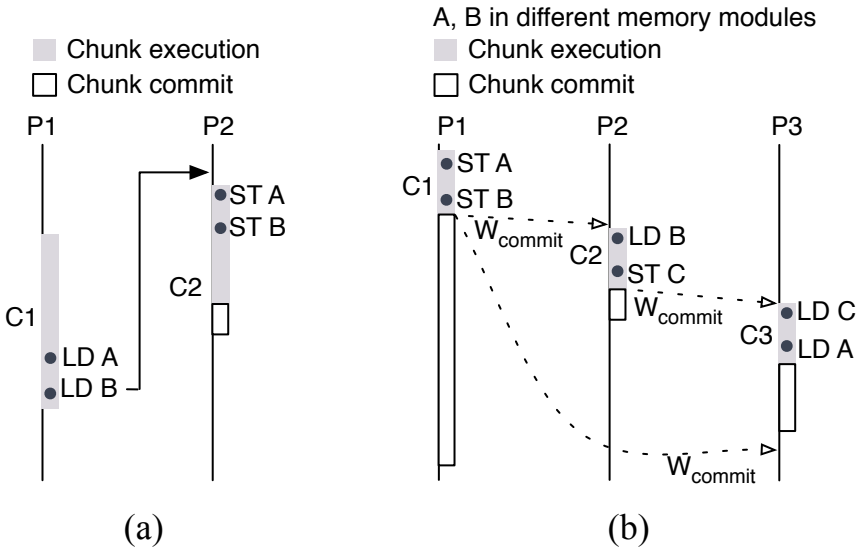


Figure 6.3: Examples of access reordering.

We efficiently enforce *Rule2* with Bulk Disambiguation (Section 4.2). When a chunk commits, it broadcasts its write signature (W_{commit}), which includes the addresses that it has updated. Any chunk *C1* that is executing intersects W_{commit} with its own R and W signatures[1]. If the intersection is not empty (there is a *collision*), *C1* is squashed and re-executed.

Enforcing the Chunk Requirements of SC. Given two consecutive chunks C_{pred} and C_{succ} from an individual processor, to enforce *CReq1* (per-processor program order requirement), we need to support two operations. The first one is to commit C_{pred} and C_{succ} in

[1]Given that a store updates only part of a cache line, W_{commit} is intersected with W to ensure that partially-updated cache lines are merged correctly (Section 4.2).

order. Note that this does not preclude the processor from *overlapping* the execution of C_{pred} and C_{succ}. Such overlapped execution improves performance, and can be managed by separately disambiguating the two chunks with per-chunk R and W signatures. However, if C_{pred} needs to be squashed, then we also squash C_{succ}.

The second operation is to update the R signature of C_{succ} correctly and in a timely manner, when there is data forwarding from a write in C_{pred} to a read in C_{succ}. In particular, the timing aspect of this operation is challenging, since the update to the R signature of C_{succ} may take a few cycles and occur after the data is consumed. This opens a window of vulnerability where a data collision between an external, committing chunk and C_{succ} could be missed: between the time a load in C_{succ} consumes the data and the time the R signature of C_{succ} is updated. Note that if C_{pred} has not yet committed, a data collision cannot be missed: the W signature of C_{pred} will flag the collision and both C_{pred} and C_{succ} will be squashed. However, if C_{pred} has committed and cleared its W signature, we can only rely on the R signature of C_{succ} to flag the collision. Consequently, the implementation must make sure that, by the time C_{pred} commits, its forwards to C_{succ} have been recorded in the R signature of C_{succ}.

CReq2 (single sequential order requirement) can be conservatively enforced with the combination of two operations: (i) *total order* of chunk commits and (ii) *atomic* chunk commit. To support the first one, we rely on the arbiter. Before a processor can commit a chunk, it asks permission to the arbiter. The arbiter ensures that chunks commit one at a time, without overlap. If no chunk is currently committing, the arbiter grants permission to the requester. To support atomic chunk commit, we disable access to all the memory locations that have been modified by the chunk, while the chunk is committing. No reads or writes from any processor to these locations in memory are allowed. When the committing chunk has made all its updates visible (e.g., by invalidating all the corresponding lines from all other caches), access to all these memory locations is re-enabled in one shot.

Advanced Design

To enforce *CReq2*, the naive design places two unnecessary constraints that limit parallelism: (i) chunk commits are completely serialized and (ii) access to the memory locations written by a committing chunk is disabled for the duration of the whole commit process. We can eliminate these constraints and still enforce *CReq2* with a simple solution: when a processor sends to the arbiter a permission-to-commit request for a chunk, it includes the chunk's R and W signatures.

To relax the first constraint, we examine the sufficient conditions for an implementation of SC at the memory access level. According to [1], the single sequential order requirement (*Req2* in Section 6.1) requires only that (i) writes to the same location be made visible to all processors in the same order (write serialization), and (ii) the value of a write not be returned by a read until the write becomes visible to all processors. Therefore, *Req2* puts constraints on the accesses to a single updated location. In a chunk environment, these constraints apply to all the locations updated by the committing chunk, namely those in its W signature. Consequently, it is safe to overlap the commits of two chunks with non-overlapping W signatures — unless we also want to relax the second constraint above (i.e., disabling access to the memory locations written by the committing chunk, for the duration of the commit), which we consider next.

To relax this second constraint, we proceed as follows. As a chunk commits, we re-enable access to individual lines gradually, as soon as they have been made visible to all other processors. Since these lines are now part of the committed state, they can be safely observed. Relaxing this constraint enhances parallelism, since it allows a currently-running chunk to read data from a currently-committing one sooner. Moreover, in a distributed-directory machine, it eliminates the need for an extra messaging step between the directories to synchronize the global end of commit.

However, there is one corner case that we must avoid because it causes two chunks to

commit out of order and, if a third one observes it, we broke *CReq2*. This case occurs if chunk *C1* is committing, chunk *C2* reads a line committed by *C1*, and then *C2* starts committing as well and finishes before *C1*. One example is shown in Figure 6.3(b). In the figure, *C1* wrote A and B and is now committing. However, A and B are in different memory modules, and while B is quickly committed and access to it re-enabled, the commit signal has not yet reached A's module (and so access to A is not disabled yet). *C2* reads the committed value of B, stores C and commits, completing before *C1*. This out-of-order commit is observed by *C3*, which reads the new C and the *old* A and commits — before receiving the incoming W signature of *C1*. We have violated SC.

A simple solution that guarantees avoiding this and similar violations is for the arbiter to deny a commit request from a chunk whose R signature overlaps with the W signature of any of the currently-committing chunks.

To summarize, in our design, the arbiter keeps the W signatures of all the currently-committing chunks. An incoming commit request includes a R and a W signature. The arbiter grants permission only if all its own W signatures have an empty intersection with the incoming R and W signature pair. This approach enables high parallelism because (i) multiple chunks can commit concurrently and (ii) a commit operation re-enables access to different memory locations as soon as possible to allow incoming reads to proceed. The latter is especially effective across directory modules in a distributed machine.

Overall *BulkSC* System

We propose to support bulk enforcement of SC as described with an architecture called *BulkSC*. *BulkSC* leverages a cache hierarchy with support for Bulk operations and a processor with efficient checkpointing. The memory subsystem is extended with an arbiter module. For generality, we focus on a system with a distributed directory protocol and a generic network. Figure 6.4 shows an overview of the architecture.

All processors repeatedly (and only) execute chunks, separated by checkpoints. As a

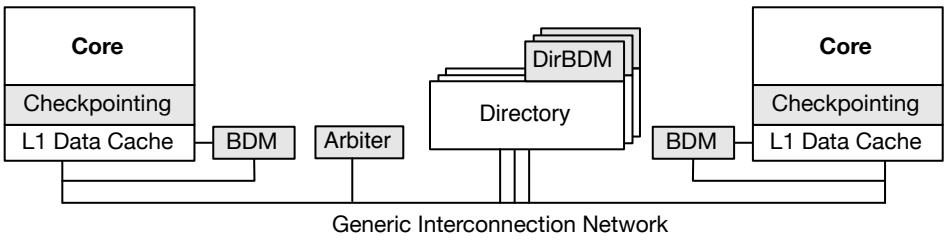


Figure 6.4: Overview of the *BulkSC* architecture.

processor executes a chunk speculatively, it buffers the updates in the cache and generates a R and a W signature in the BDM. When chunk i completes, the processor sends a request to commit to the arbiter with signatures R_i and W_i. The arbiter intersects R_i and W_i with the W signatures of all the currently-committing chunks. If all intersections are empty, W_i is saved in the arbiter and also forwarded to all interested directories for commit. Individual directories use a DirBDM module to perform signature expansion (Seciotn 3.3) on W_i to update their sharing state, and forward W_i to interested caches. The BDM in each cache uses W_i to perform bulk disambiguation and potentially squash local chunks. Memory accesses within a chunk are fully overlapped and reordered, and an individual processor can overlap the execution of multiple chunks.

6.2.3 Interaction with Explicit Synchronization

A machine with *BulkSC* runs code with explicit synchronization operations correctly. Such operations are executed inside chunks. While they have the usual semantics, they neither induce any fences nor constrain access reordering within the chunk in any way.

Figure 6.5 shows some examples with lock acquire and release. In Figure 6.5(a), acquire and release end up in the same chunk. Multiple processors may execute the critical section concurrently, each assuming that it owns the lock. The first one to commit the chunk squashes the others. The longer the chunk is relative to the critical section, the higher the potential for hurting parallelism is — although correctness is not affected. Figure 6.5(b)

shows a chunk that includes two critical sections. Again, this case may restrict parallelism but not affect correctness. In general, the longer the chunk size is, the higher the chance of hurting parallelism. Finally, Figure 6.5(c) shows a chunk that only contains the acquire. Multiple processors may enter the critical section believing they own the lock. However, the first one to commit squashes the others which, on retry, find the critical section busy.

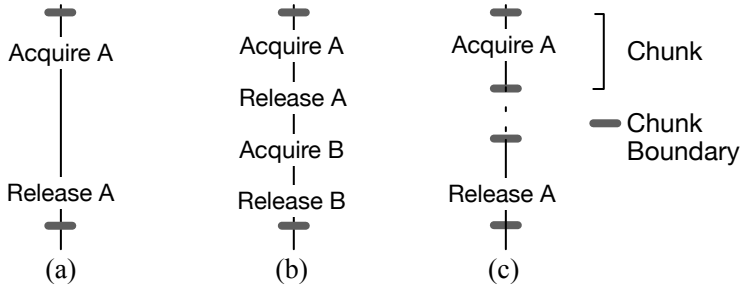


Figure 6.5: Interaction of *BulkSC* with explicit synchronization.

Similar examples can be constructed with other primitives. A worst case occurs when all processors but one are waiting on a synchronization event — e.g., when they are waiting on the processor that holds a lock, the processor that arrives last to a barrier, or the one that will set a flag. In this case, *BulkSC* guarantees that the key processor makes forward progress. To see why, note that the waiting processors are spinning on a variable. Committing a chunk that just reads a variable cannot squash another chunk. However, one could envision a scenario where the spin loop includes a write to variable v, and v (or another variable in the same memory line) is read by the key processor. In this case, the latter could be repeatedly squashed. Note that these observations are oblivious to the low-level synchronization instructions supported by the hardware, such as *load-linked/store-conditional*.

This problem is avoided by dynamically detecting when a chunk is being repeatedly squashed and then taking a measure to prevent future squashing. *BulkSC* includes two such measures: one for high performance that works in the common case, and one that is slow but guarantees progress. The first one involves exponentially decreasing the size of the chunk after each squash, thereby significantly increasing the chances that the chunk will commit.

However, even reducing a chunk's size down to a single write does not guarantee forward progress. The second measure involves pre-arbitrating. Specifically, the processor first asks the arbiter permission to execute; once permission is granted, the processor executes while the arbiter rejects commit requests from other processors. After the arbiter receives the commit request from the first processor, the system returns to normal operation.

Finally, while chunks and transactions share some similarities, chunks are not programming constructs like transactions. Indeed, while transactions have static boundaries in the code, chunks are dynamically built by the hardware from the dynamic instruction stream. Therefore, they do not suffer the livelock problems pointed out by Blundell *et al.* [10].

6.3 BulkSC Architecture

We consider the three components of the *BulkSC* architecture: support in the processor and cache (Section 6.3.1), the arbiter module (Section 6.3.2), and directory modifications (Section 6.3.3).

6.3.1 Processor and Cache Architecture

Chunk Execution

Processors dynamically break the instruction stream into chunks, creating a checkpoint at the beginning of each chunk. As indicated before, within-chunk execution proceeds with all the memory access reordering and overlapping possible in uniprocessor code. Explicit synchronization instructions do not insert any fences or constrain access reordering in any way. In addition, a processor can have multiple chunks in progress, which can overlap their memory accesses. As chunks from other processors commit, they disambiguate their accesses against local chunks, which may lead to the squash and re-execution of a local chunk. Finally, when a chunk completes, it makes all its state visible with a commit.

This mode of execution is efficiently underpinned by the mechanisms of Bulk (Chapter 4) and of checkpointed processors [3, 14, 21, 42, 43, 50, 57, 65]. Bulk enables the inexpensive recording of the addresses accessed by a chunk in a R and W signature in the BDM. Loads update the R signature when they bring data into the cache, while stores update the cache and the W signature when they reach the ROB head — even if there are other, in-progress stores. Forwarded loads also update the R signature. With such signatures, chunk commit involves sending the chunk's R and W signatures to the arbiter and, on positive reply, clearing them. Cross-chunk disambiguation involves intersecting the incoming W signature of a committing chunk against the local R and W signatures — and squashing and re-executing the chunk if the intersection is not empty. Chunk rollback leverages the mechanisms of checkpointed processors: a register checkpoint is restored and all the speculative state generated by the chunk is efficiently discarded from the cache.

With this design, there is no need to snoop the load-store queue to enforce consistency, or to have a history buffer as in [30]. An SC violation is detected when a bulk disambiguation operation detects a non-empty intersection between two signatures.

Moreover, there is no need to watch for cache displacements to enforce consistency. Clean lines can be displaced from the cache, since the R signature records them, while the BDM prevents the displacement of speculatively written lines until commit (Section 4.5). Thanks to the BDM, the cache tag and data array are unmodified; they do not know if a given line is speculative or what chunk it belongs to.

Chunk Duration and Multiple-Chunk Support

The processor uses instruction count to decide when to start a new chunk. While chunks should be large enough to amortize the commit cost, very large chunks could suffer more conflicts. In practice, performance is fairly insensitive to chunk size, and we use chunks of $\approx$1,000 instructions. However, if the processor supports checkpoint-based optimizations, such as resource recycling [3, 50] or memory latency tolerance [14, 43, 65], it would make

sense to use their checkpoint-triggering events as chunk boundaries. Finally, a chunk also finishes when its data is about to overflow a cache set.

A processor can have multiple chunks in progress. For this, it leverages Bulk's support for multiple pairs of signatures and a checkpointed processor's ability to have multiple outstanding checkpoints. When the processor decides that the next instruction to rename starts a new chunk, it creates a new checkpoint, allocates a new pair of R and W signatures in the BDM, and increments a set of bits called *Chunk ID*. The latter are issued by the processor along with every memory address to the BDM. They identify the signature to update.

Instructions from multiple local chunks can execute concurrently and their memory accesses can be overlapped and reordered, since they update different signatures. An incoming W signature performs disambiguation against all the local signature pairs and, if a chunk needs to be squashed, all its successors are also squashed.

Before a chunk can start the commit process, it ensures that all its forwards to local successor chunks have updated the successors's R signatures. As indicated in Section 6.2.2, this is required to close a window of vulnerability due to the lag in updating signatures. In our design, on any load forwarding, we log an entry in a buffer until the corresponding R signature is updated. Moreover, a completed chunk cannot initiate its commit arbitration until it finds the buffer empty. While a chunk arbitrates for commit or commits, its local successor chunks can continue execution. Local chunks must request and obtain permission to commit in strict sequential order. After that, their commits can overlap.

I/O

I/O and other uncached operations cannot be executed speculatively or generally overlapped with other memory accesses. When one such instruction is renamed, the processor stalls until the current chunk completes its commit — checking this event may require inspecting the arbiter. Then, the operation is fully performed. Finally, a new chunk is started. To support these steps, we reuse the relevant mechanisms that exist to deal with I/O in checkpointed

processors.

Summary: Simplicity and Performance

We claim that *BulkSC*'s SC implementation is simple because it largely *decouples* memory consistency enforcement from processor structures. Specifically, there is no need to perform associative lookups of sizable structures in the processor, or to interact in a tightly-coupled manner with key structures such as the load-store queue, ROB, register file, or map table.

This is accomplished by leveraging Bulk and checkpointed processors. With Bulk, detection of SC violations is performed with simple signature operations outside the processor core. Additionally, caches are oblivious of what data is speculative (their tag and data arrays are unmodified), and do not need to watch for displacements to enforce consistency. Checkpointing provides a non-intrusive way to recover from consistency-violating chunks.

In our opinion, this decoupling enables designers to conceive both simple and aggressive (e.g., CFP [65], CAVA/Clear [14, 43], CPR [3] or Kilo-instruction [21]) processors with much less concern for memory consistency issues.

BulkSC delivers high performance by allowing any reordering and overlapping of memory accesses within a chunk. Explicit synchronization instructions induce no fence or reordering constraint. Moreover, a processor does not stall on chunk transitions, and it can overlap and reorder memory operations from different chunks. Finally, arbitration for chunk commit is quick. It is not a bottleneck because it only requires quick signature intersections in the arbiter. In the meantime, the processor continues executing.

6.3.2 Arbiter Module

Baseline Design

The arbiter is a simple state machine whose role is to enforce the minimum serialization requirements of chunk commit. The arbiter stores the W signatures of all the currently-

committing chunks. It receives permission-to-commit requests from processors that include the R and W signatures of a chunk. The arbiter takes each of the W signatures in its list and intersects them with the incoming R and W signatures. If any intersection is not empty, permission is denied; otherwise, it is granted, and the incoming W signature is added to the list. Since this process is very fast, the arbiter is not a bottleneck in a modest-sized machine. A processor whose request is denied will later retry.

Figure 6.6(a) shows the complete commit process. Processor A sends a permission-to-commit message to the arbiter, together with R and W signatures (1). Based on the process just described, the arbiter decides if commit is allowed. It then sends the outcome to the processor (2) and, if the outcome was positive, it forwards the W signature to the relevant directories (2). Each directory processes W (Section 6.3.3) and forwards it to the relevant processors (3), collects operation-completion acknowledgments (4) and sends a completion message to the arbiter (5). When the arbiter receives all the acknowledgments, it removes the W signature from its list.

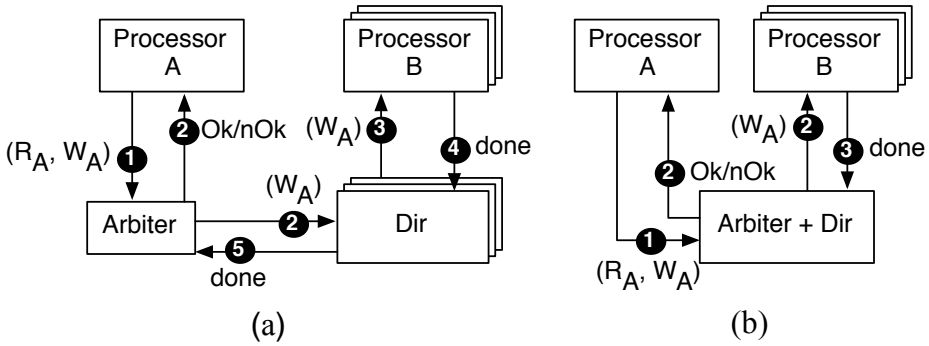


Figure 6.6: Commit process with separate (a) and combined (b) arbiter and directory.

Note that a processor whose permission-to-commit request was denied will not necessarily be squashed. Consider the case when a processor R is building a chunk and suffers a cache miss. The miss is serviced by a processor C that was granted permission to commit but has not finished committing yet. R might have received C's write signature W_C before it suffered the cache miss. R will fetch the new data from C and attempts to commit before

C's commit is complete. The arbiter will deny R's permission-to-commit request but R will not receive W_C again and therefore might not suffer a squash before trying to commit again.

In a small machine, such as a few-core chip multiprocessor, there may be a single directory. In this case, the arbiter can be combined with it. The resulting commit transaction is shown in Figure 6.6(b).

RSig Commit Bandwidth Optimization

The list of W signatures in the arbiter of a modest-sized machine is frequently empty. This is because the commit process is fast and — as we will see in Section 6.4 — chunks that only write thread-private data have an empty W signature.

When the arbiter's W signature list is empty there are no active commits in progress. In such case, the arbiter has no use for the R signature included in a permission-to-commit request. We can save network bandwidth if we do not include the R signature in the message. Consequently, we improve the commit design by always sending only the W signature in the permission-to-commit request. In the frequent case when the arbiter's list is empty, the arbiter grants permission immediately; otherwise, it requests the R signature from the processor and proceeds as before. We call this optimization *RSig* and include it in the baseline *BulkSC* system.

Distributed Arbiter

In large machines, the arbiter may be a bottleneck. To avoid this case, we distribute the arbiter into multiple modules, each managing a range of addresses. An arbiter now only receives commit requests from chunks that have accessed its address range (plus potentially other ranges as well). To commit a chunk that only accessed a single address range, a processor only needs to communicate with one arbiter. For chunks that have accessed multiple ranges, multiple arbiters need to be involved. In this case, each arbiter will make a decision based on the partial information that it has — the W signatures of the committing

chunks that have written its address range. We then need an extra arbiter module that coordinates the whole transaction. We call this module *Global Arbiter* or *G-arbiter*.

Figure 6.7 shows the two possible types of commit transactions, in a machine where we have distributed the arbiter with the distributed directory. Figure 6.7(a) shows the common case of a commit that involves a single arbiter. The processor knows from the signatures which arbiter to contact. Figure 6.7(b) shows the case when multiple arbiters are involved. In this case, the processor sends the signatures to the G-arbiter (1), which forwards them to the relevant arbiters (2). The arbiters check their W list and send their responses to the G-arbiter (3), which combines them and informs all parties (4). The transaction then proceeds as usual.

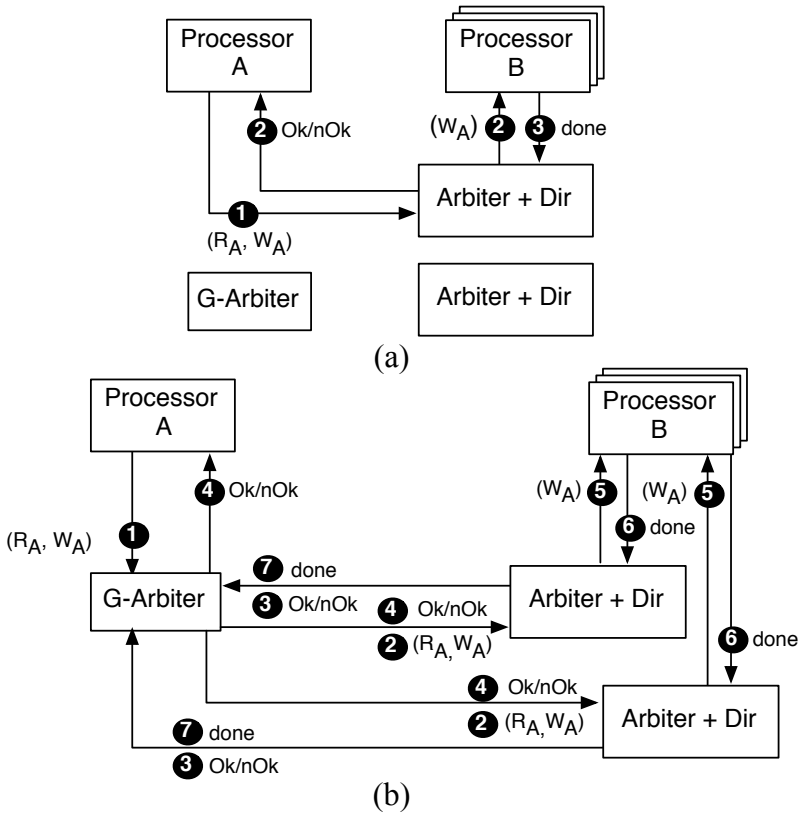


Figure 6.7: Distributed arbiter with a commit that involves a single arbiter (a) or multiple (b).

When the G-arbiter is used, the commit transaction has longer latency and more mes-

sages. We can speed up some transactions by storing in the G-arbiter the W signatures of all the currently-committing chunks whose requests went through the G-arbiter. With this, we are replicating information that is already present in some arbiters, but it may help speed up transactions that are denied. Indeed, in Figure 6.7(b), when the G-arbiter receives message (1), it checks its W list for collisions. If it finds one, it immediately denies permission.

6.3.3 Directory Module

BulkSC does not require a machine with a broadcast link to work. For generality, this thesis presents a design with distributed memory and directory. However, we need to extend the directory to work with the inexact information of signatures. This is done by enhancing each directory module with a module called *DirBDM* that supports basic bulk operations (Figure 6.4). When a directory module receives the W signature of a committing chunk, the DirBDM performs three operations: (i) expand the signature into its component addresses and update the directory state, (ii) based on the directory state, forward W to the relevant caches for address disambiguation, and (iii) conservatively disable access to the directory entries of all the lines written by the committing chunk in this module, until the new values of the lines are visible to all processors — i.e., until the old values of the lines have been invalidated from all caches.

Operation (iii) is conservative; we could have re-enabled access to individual lines in the directory module progressively, as they get invalidated from all caches. However, we choose this implementation for simplicity. Still, different directory modules re-enable access at different times.

In our discussion, we use a full bit-vector directory [46] for simplicity. Directory state can be updated when a chunk commits, when a non-speculative dirty line is written back, or when the directory receives a demand cache miss from a processor. The latter are always read requests — even in the case of a write miss — and the directory adds the requester processor as a sharer. The reason is that, because the access is speculative, the directory

cannot mark the requester processor as keeping an updated copy of the line.

Signature Expansion

On reception of a W signature, the DirBDM performs a signature-expansion bulk operation (Section 3.2) to determine what entries in the directory structure may have their addresses encoded in the signature. For each of these entries, it checks the state and, based on it, (i) compiles a list of processors to which W will be forwarded for bulk disambiguation and (ii) updates the entry's state. The list of processors that should receive W is called the Invalidation List.

A key challenge is that the signature expansion of W may produce the address of lines that have not been written. Fortunately, a careful analysis of all possible cases ensures that the resulting directory actions never lead to incorrect execution. To see why, consider the four possible states that one of the selected directory entries can be at (Table 6.1). In all cases, we may be looking at a line that the chunk did not actually write.

In the table, cases 1 and 3 are clearly false positives: if the committing chunk had written the line, its processor would have accessed the line and be recorded in the bit vector as a sharer, already. Therefore, no action is taken. Case 4 requires no action even if it is not a false positive. Case 2 requires marking the committing processor as keeping the only copy of the line, in state dirty, and adding the rest of current sharer processors to the Invalidation List. If this is a false positive, we are incorrectly changing the directory state and maybe forwarding W to incorrect processors. The latter can at most cause unnecessary (and rare) chunk squashes — it cannot lead to incorrect execution; the former sets the coherence protocol to a state that many existing cache coherence protocols are already equipped to handle gracefully — consequently, it is not hard to modify the protocol to support it. Specifically, it is the same state that occurs when, in a MESI protocol [59], a processor reads a line in Exclusive mode and later displaces it from its cache silently. The directory thinks that the processor owns the line, but the processor does not have it.

Case	Current Entry State		Action	Notes
	Dirty Bit Set?	Committing Proc is in Bit Vector?		
1	No	No	Do nothing	False positive. Committing proc should have accessed the data and be in bit vector already
2	No	Yes	1) Add sharer procs to Invalidation List 2) Reset rest of bit vector 3) Set Dirty bit	Committing proc becomes the owner
3	Yes	No	Do nothing	False positive. Committing proc should have accessed the data and be in bit vector already
4	Yes	Yes	Do nothing	Committing proc already owner

Table 6.1: Possible states of a directory entry selected after signature expansion and action taken.

In our protocol, at the next cache miss on the line by a processor, the directory will ask the "false owner" for a writeback. The latter will respond saying it does not have a dirty copy. The directory will then provide the line from memory, and change the directory state appropriately.

Disabling Access from Incoming Reads

As per Section 6.2.2, *CReq2* (single sequential order requirement) requires that no processor use the directory to see the new value of a line from the committing chunk, before all processors can see the new value. As indicated above, our conservative implementation disables reads to any line updated by the committing chunk in this module, from the time the directory module receives the W signature from the arbiter (Message (2) in Figure 6.6(a)) until it receives the "done" messages from all caches in the Invalidation List (Messages (4)).

This is easily done with bulk operations. The DirBDM intercepts all incoming reads and applies the *membership* bulk operation (Section 3.2) to them, to see if they belong to W. If they do, they are bounced.

Note that the directory is not completely unavailable during an expansion, only the lines involved in the expansion are unavailable to incoming reads. Also as our characterization will show in Section 6.6.3, typically very few entries are updated during expansion. Because of these two reasons, the performance impact of blocking incoming reads during expansion is likely to be low.

Directory Caching

In *BulkSC*, using directory caches [32] is preferred over full-mapped directories because they limit the number of false positives by construction. With a directory cache, the DirBDM uses signature expansion (Section 3.2) on incoming W signatures with a different decode (δ) function than for the caches — since directories have different size and associativity.

However, directory caches suffer entry displacements. In conventional protocols, a displacement triggers the invalidation of the line from all caches and, for a dirty line, a write-back. In *BulkSC*, a conservative approach is to additionally squash all the currently-running chunks that may have accessed the line. Consequently, when the directory displaces an entry, it builds its address into a signature and sends it to all sharer caches for bulk disambiguation with their R and W signatures. This operation may squash chunks, and will invalidate (and if needed write back) any cached copies of the line.

This protocol works correctly for a chunk that updated the displaced line and is currently committing. Since the chunk has already cleared its signatures, disambiguation will not squash it. Moreover, since the line is dirty non-speculative in the processor's cache, the protocol will safely write back the line to memory.

6.3.4 Putting It All Together: The Complete Commit Process

This section describes the complete commit process, from the moment a processor completes a chunk until it is visible to all processors in the system. The process described here includes the *RSig* optimization discussed in Section 6.3.2.The committing processor will be named C and R represents all processors in the system that observe that commit.

When processor C completes a chunk and is ready to commit, it follows the flowchart shown in Figure 6.8. C requests commit permission from the arbiter by sending it a request that includes its write signature W_C. If the arbiter also needs R_C for the arbitration, according the *RSig* optimization, it will request it from C. If the arbiter grants commit permission, the processor commits the chunk by clearing W_C and R_C (see Section 4.1).

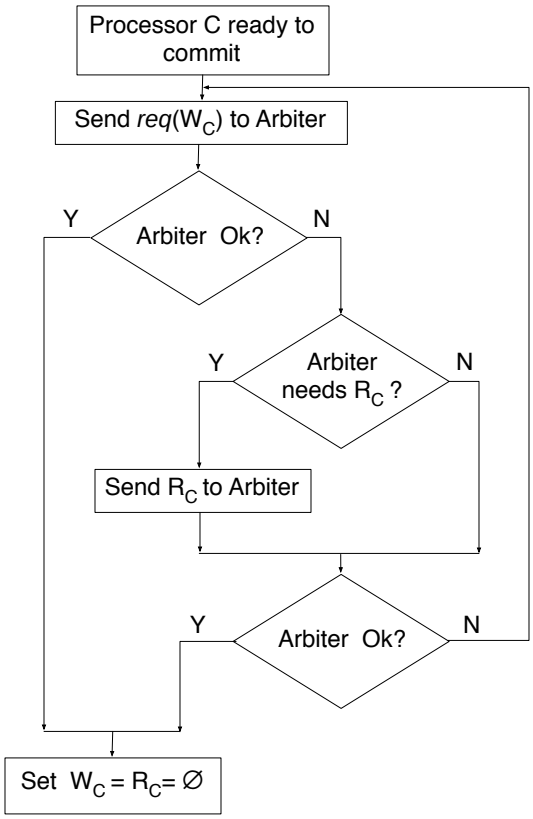


Figure 6.8: Flowchart of the commit process in the committing processor.

The arbiter keeps a list of write signatures that belong to chunks in the process of

being committed. This structure is called the Pending List. As Figure 6.9 illustrates, upon reception of W_C, if the Pending List is empty, the arbiter grants commit permission immediately. If the Pending List is not empty, the arbiter requests R_C from processor C and then determines if W_C or R_C overlap with any signature in the Pending List. If so, it denies C's commit request. When commit permission is granted, if W_C is empty, the arbiter simply sends the *ok* to processor C. If W_C is not empty, the arbiter needs to insert it in the Pending List and send it to the relevant directories. When all directories send their *ack* to the arbiter, W_C is removed from the Pending List. Section 6.3.2 presents more details on the arbiter design including how to design distributed arbiters.

Figure 6.10 shows the actions taken by the DirBDM when the directory receives a signature from the arbiter. When a directory receives W_C from the arbiter, the DirBDM performs a signature expansion within the directories entries (see Section 3.3 and Section 6.3.3). As described in Section 6.3.3, for each resulting entry, if the line is dirty and C is a sharer, the corresponding sharers are added to the Invalidation List, the line's owner is set to C and the dirty bit is set to 1. The Invalidation List, which is initially empty, holds the information of what nodes need to receive the signature after the expansion is over. For the paths annotated with "*" no action is taken, as Table 6.1 shows, this happen either because it is guaranteed to be a false positive in the expansion or C is already the owner of the line. After all entries that resulted from the expansion are processed, the DirBDM sends W_C to all processors in the Invalidation List and waits for them to acknowledge its reception. After all *acks* are received, the DirBDM acks the arbiter. During expansion, all lines subject to an action are blocked for read accesses, Figure 6.11 illustrates how the directory determines if an access to a line should be blocked.

Figure 6.12 shows the actions taken when a processor receives W_C from the directory. There are two main actions, disambiguation, which determines if a squash is necessary due to a potential consistency violation; and invalidation, which guarantees that data in the cache is kept coherent.

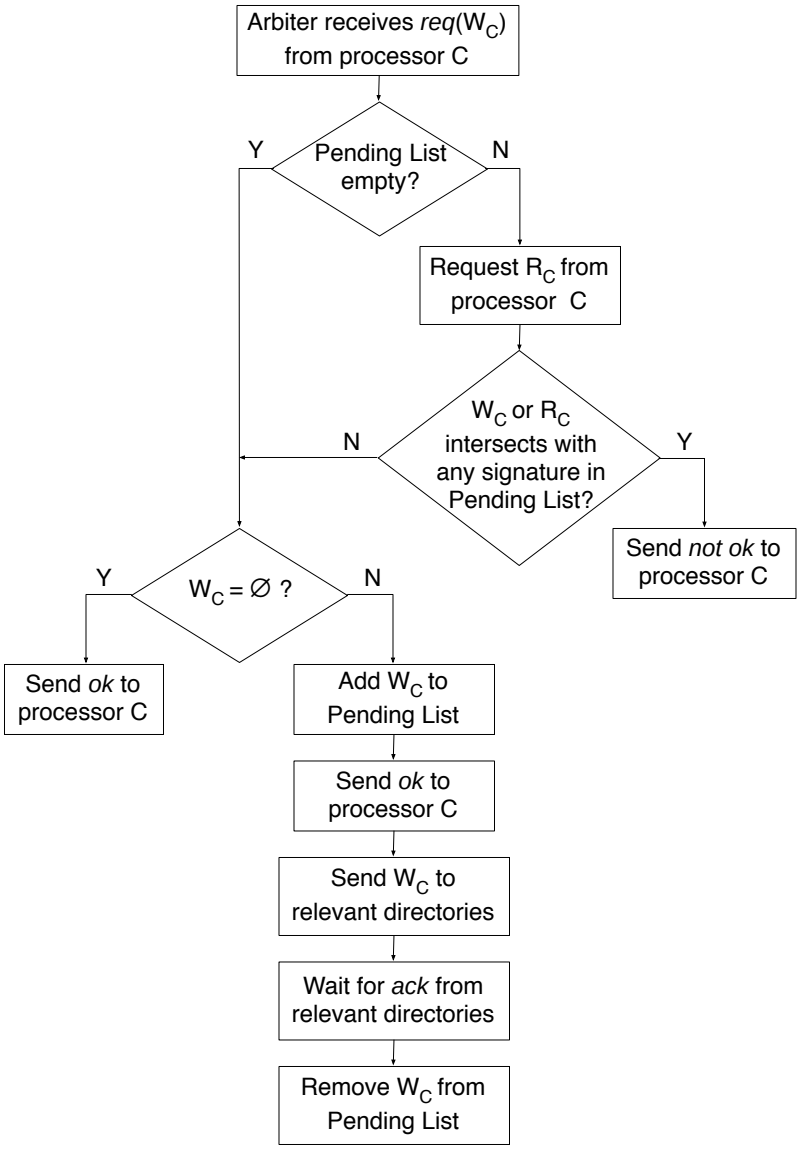


Figure 6.9: Flowchart of the arbitration process.

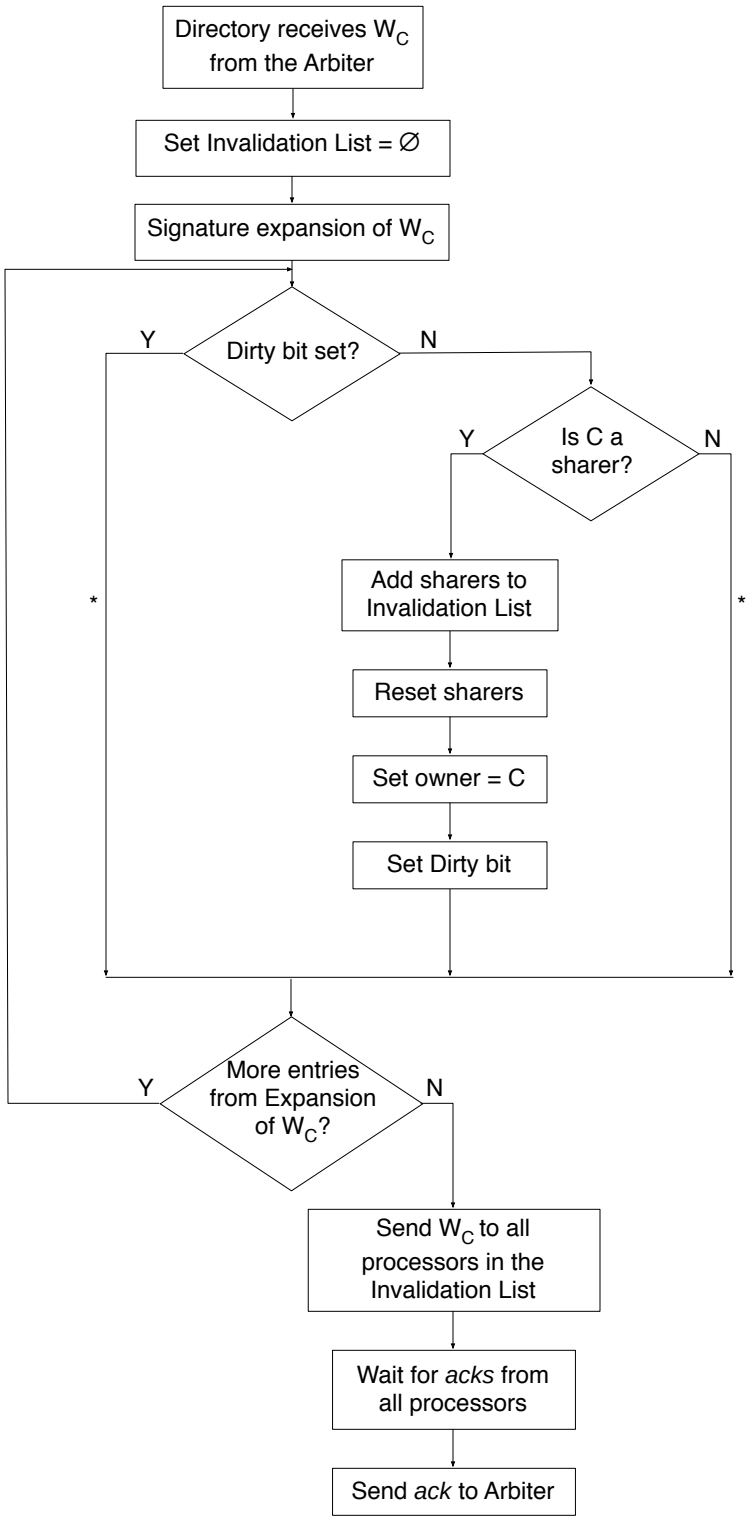


Figure 6.10: Flowchart of the actions taken by the DirBDM when it receives a signature from the Arbiter.

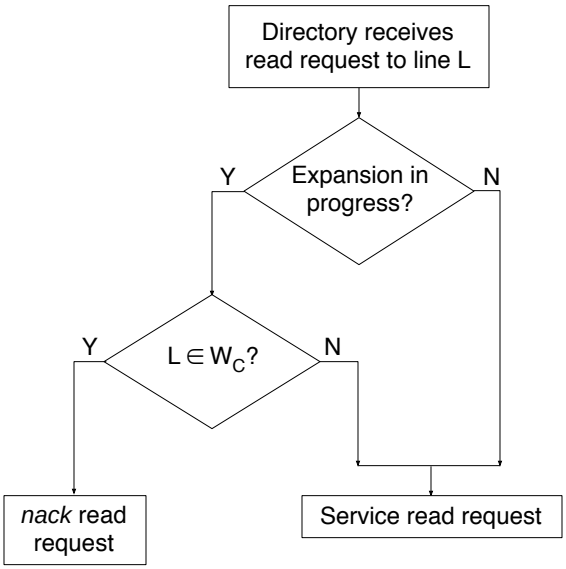


Figure 6.11: Flowchart of how the directory services a read.

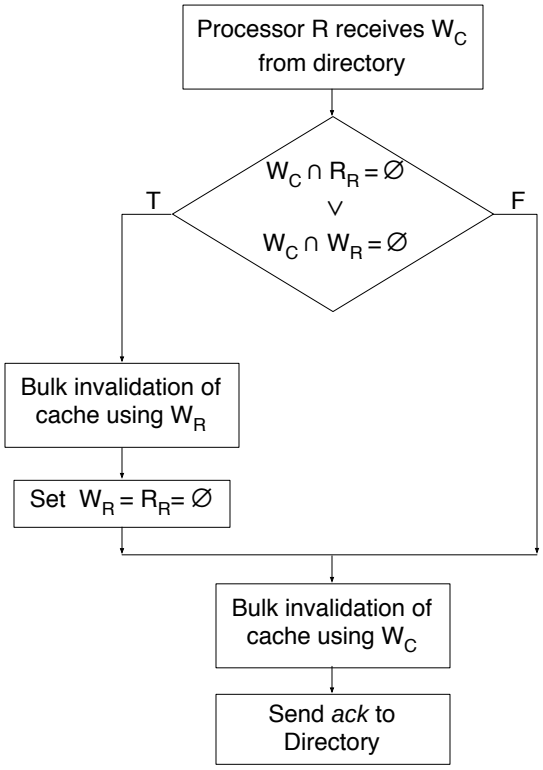


Figure 6.12: Flowchart of the process of receiving a signature from a committing processor.

6.4 Leveraging Private Data

Accesses to private data are not subject to consistency constraints. Consequently, they do not need to be considered when disambiguating chunks or arbitrating for commits — they can be removed from a chunk's W signature.

Removing writes to private data from W also has the substantial benefit of reducing signature pollution. This decreases aliasing and false positives, leading to fewer unnecessary cache invalidations and chunk squashes. Another benefit is that the resulting W is often empty. This reduces the number of entries in the arbiter's list of W signatures — since a permission-to-commit request with a zero W does not update the list. As a result, such a list is more often empty. This has enabled the *RSig* commit bandwidth optimization of Section 6.3.2.

To handle private data, we propose to include in the BDM an additional signature per running chunk called W_{priv}. Writes to private data update W_{priv} rather than W. W_{priv} is used neither for bulk disambiguation nor for commit arbitration. In the following, we present two schemes that use W_{priv} to leverage either statically-private data or dynamically-private data. The latter is data that, while perhaps declared shared, is accessed by only one processor for a period of time. These two schemes use W_{priv} differently.

6.4.1 Leveraging Statically-Private Data

This approach relies on the software to convey to the hardware what data structures or memory regions are private. A simple implementation of this approach is to have a page-level attribute checked at address-translation time that indicates whether a page contains private or shared data. On a read to private data, the R signature is not updated, thereby avoiding polluting R; on a write to private data, W_{priv} is updated rather than W.

To commit a chunk, the processor only sends W to the arbiter. Once commit permission is granted, the processor sends W_{priv} directly to the directory for signature expansion, so

that private data is kept *coherent*. Coherence of private data is necessary because threads can migrate between processors, taking their private data to multiple processors. With this approach, we have divided the address space into a section where SC is enforced and another where it is not.

6.4.2 Leveraging Dynamically-Private Data

In many cases, a processor P updates a variable v in multiple chunks without any intervening access to v from other processors. Using our protocol of Section 6.3, every first write of P to v in a chunk would require the update of W and the writeback of v's line to memory. The latter is necessary because v's line is in state dirty non-speculative before the write.

We want to optimize this common case by (i) skipping the writeback of v's line to memory at every chunk and (ii) not polluting W with the writes to v. Our optimization, intuitively, is to update v keeping its line in dirty *non-speculative* state in the cache. If, by the time the chunk completes, no external event required the old value of v, the processor commits the chunk without informing the arbiter or directory that the chunk updated v — more specifically, the processor sends to the arbiter a W signature that lacks the writes to v.

To support this, we make two changes to *BulkSC*. First, every time that a processor writes to a line that is dirty non-speculative in its cache, the hardware updates W_{priv} rather than W, and does not write the line back to memory. These lines can be easily identified by the hardware: they have the Dirty bit set and their address is not in W. With this change, the commit process will not know about the updates to these lines.

The second change is that, the first time that these lines are updated in a chunk, the hardware also saves the version of the line before the update in a small *Private Buffer* in the BDM, while the updated version of the line is kept in the cache. The hardware easily identifies that this is the first update: the Dirty bit set and the line's address is neither in W nor in W_{priv}. We save the old value of the line in this buffer in case it is later required. This buffer can hold ≈24 lines and is not in any critical path.

With this support, when a chunk is granted permission to commit based on its W, the hardware clears the Private Buffer and W_{priv}. We have skipped the writeback of the lines in the buffer.

There are, however, two cases when the old value of the line is required. The first one is when the chunk gets squashed in a bulk disambiguation operation. In this case, the lines in the Private Buffer are copied back to the cache — and the Private Buffer and W_{priv} are cleared.

The second case is when our predicted private pattern stops working, and another processor requests a line that is in the Private Buffer. This is detected by the BDM, which checks every external access to the cache for membership ($\in$) in W_{priv}. This is a fast bulk operation (Section 3.2). In the (unusual) case of a non-empty result, the Private Buffer is checked. If the line is found, it is supplied from there rather than from the cache, and the address is added to W. Intuitively, we have to provide the old copy of the line and "add back" the address to W. Similarly, if a line overflows the Private Buffer, it is written back to memory and its address is added to W.

6.5 Discussion

Past approaches to supporting high-performance SC (Section 6.1) add mechanisms local to the processor to enforce consistency; *BulkSC*, instead, relies on an external arbiter. We argue, however, that this limitation is outweighed by *BulkSC*'s advantage of simpler processor hardware — coming from decoupling memory consistency enforcement from processor microarchitecture and eliminating associative lookups (Section 6.3.1).

In addition, the amount of commit bandwidth required by *BulkSC* is very small. There are three reasons for this. First, since chunks are large, the few messages needed per individual commit are amortized over many instructions. Second, we have presented optimizations to reduce the commit bandwidth requirements due to R (Section 6.3.2) and W (Section 6.4)

signatures; these optimizations are very effective, as we will show later. Finally, in large machines with a distributed arbiter, if there is data locality, commits only access a local arbiter.

BulkSC's scalability is affected by two main factors: the ability to provide scalable arbitration for chunk commit, and whether the superset encoding used by signatures limits scalability in any way. To address the first factor, we have presented commit bandwidth optimizations, and a distributed arbiter design that scales as long as there is data locality. Superset encoding could hurt scalability if the longer chunks needed to tolerate longer machine latencies ended up creating much address aliasing. In practice, while we evaluated some signature configurations in Section 5.3.5, there is still a large unexplored design space of signature size and encoding. Superset encoding could also hurt scalability if it induced an inordinate number of bulk disambiguations as machine size increases. Fortunately, Section 6.3.3 showed that signature expansion can leverage the directory state to improve the precision of sharer information substantially.

Finally, write misses are a common source of stalls in multiprocessors. The effect of these misses on performance is more pronounced in stricter memory consistency models. In *BulkSC*, writes are naturally stall-free. Specifically, writes retire from the head of the reorder buffer even if the line is not in the cache — although the line has to be received before the chunk commits. Moreover, writes do not wait for coherence permissions because such permissions are implicitly obtained when the chunk is allowed to commit.

6.6 Evaluation

6.6.1 Experimental Setup

We evaluate *BulkSC* using the SESC [62] cycle-accurate execution-driven simulator with detailed models for processor, memory subsystems and interconnect. For comparison, we implement SC with hardware prefetching for reads and exclusive prefetching for writes [27].

In addition, we implement RC with speculative execution across fences and hardware exclusive prefetching for writes. The machine modeled is an 8-processor CMP with a single directory. Table 6.2 shows the configurations used.

We use 11 applications from SPLASH-2 (all but Volrend, which cannot run in our simulator), and the commercial applications SPECjbb2000 and SPECweb2005. SPLASH-2 applications run to completion. The commercial codes are run using the Simics full-system simulator as a front-end to our SESC simulator. SPECjbb2000 is configured with 8 warehouses and SPECweb2005 with the e-commerce workload. Both run for over 1B instructions after initialization.

We evaluate the *BulkSC* configurations of Table 6.2: BSC_{base} is the basic design of Section 6.3; BSC_{dypvt} is BSC_{base} plus the dynamically-private data optimization of Section 6.4.2; BSC_{stpvt} is BSC_{base} plus the statically-private data optimization of Section 6.4.1; and BSC_{exact} is BSC_{base} with an alias-free signature. We also compare them to RC, SC and SC++ [30].

6.6.2 Performance

Figure 6.13 compares the performance of BSC_{base}, BSC_{dypvt}, BSC_{exact}, and BSC_{stpvt} to that of SC, RC and SC++. In the figure, *SP2-G.M.* is the geometric mean of SPLASH-2. Our preferred configuration, BSC_{dypvt}, performs about as well as RC and SC++ for practically all applications. Consequently, we argue that BSC_{dypvt} is the most attractive design of the three, as it is simple to implement and still supports SC. The only exception is radix, which has frequent chunk conflicts due to aliasing in the signature. The performance difference between RC and SC is large, and in line with [58].

Comparing BSC_{base} and BSC_{dypvt}, we see the impact of the dynamically-private data optimization. It improves performance by 6% in SPLASH-2, 3% in SPECjbb and 11% in SPECweb. Most of the gains come from reducing the pollution in the W signature, leading to fewer line invalidations and chunk squashes. The small difference between BSC_{exact} and

Processor and Memory Subsystem		*BulkSC*
Cores: 8 in a CMP	Private writeback D-L1:	
Frequency: 5.0 GHz	size/assoc/line:	Signature:
Fetch/issue/comm width: 6/4/5	32KB/4-way/32B	Size: 2 Kbits
I-window/ROB size: 80/176	Round trip: 2 cycles	Conf: Like in Table 5.3
LdSt/Int/FP units: 3/3/2	MSHRs: 8 entries	# Chunks/Proc: 2
Ld/St queue entries: 56/56	Shared L2:	Chunk Size: 1000 inst.
Int/FP registers: 176/90	size/assoc/line:	Commit Arb. Lat.: 30 cyc
Branch penalty: 17 cyc (min)	8MB/8-way/32B	Max. Simul. Commits: 8
	Round trip: 13 cycles	# of Arbiters: 1
	MSHRs: 32 entries	# of Directories: 1
	Mem roundtrip: 300 cyc	

Configurations Used	
BSC_{base}	Basic *BulkSC* (Section 6.3)
BSC_{dypvt}	BSC_{base} + dyn. priv. data opt (Section 6.4.2)
BSC_{stpvt}	BSC_{base} + static priv. data opt (Section 6.4.1)
BSC_{exact}	BSC_{dypvt} + "magic" alias-free signature
SC	Includes prefetching for reads + exclusive prefetching for writes
RC	Includes speculative execution across fences + exclusive prefetching for writes
SC++	Model from [30] w/ 2K-entry SHiQ

Table 6.2: Simulated system configurations. All latencies correspond to an unloaded machine.

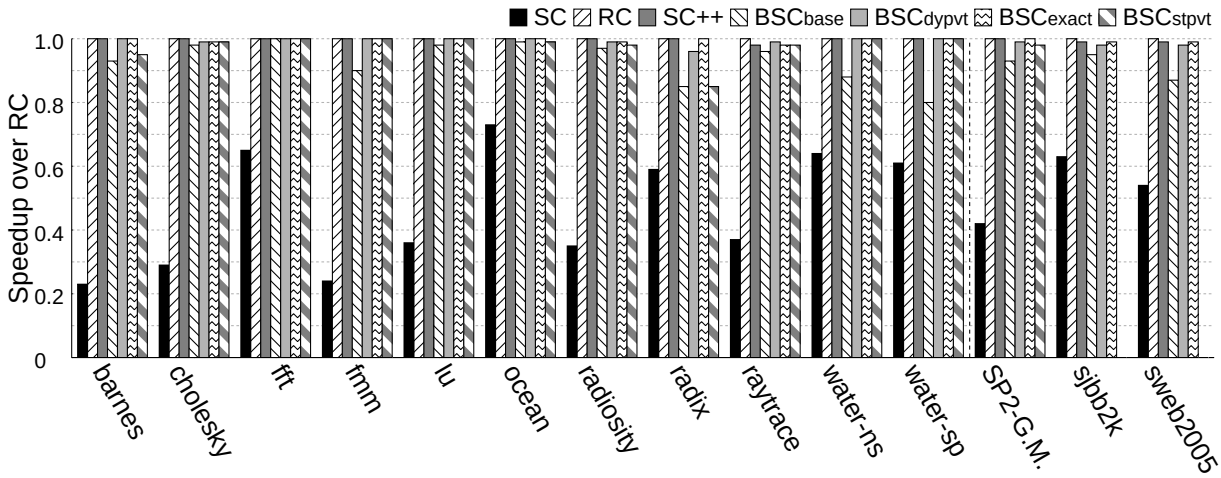


Figure 6.13: Performance of several *BulkSC* configurations, SC, RC and SC++, all normalized to RC.

BSC_{dypvt} shows that the dynamically-private data optimization reduces aliasing enough to make BSC_{dypvt} behave almost as if it had an ideal signature.

For BSC_{stpvt}, we consider all stack references as private and everything else as shared. Unfortunately, we can only apply it to SPLASH-2 applications because of limitations in our simulation infrastructure. As Figure 6.13 shows, BSC_{stpvt} improves over BSC_{base} by a geometric mean of 5%, leaving BSC_{stpvt} within 2% of the performance of BSC_{dypvt}. The only application with no noticeable benefit is radix, which has very few stack references.

Figure 6.14 shows the performance of BSC_{dypvt} with chunks of *1000*, *2000* and *4000* instructions. *4000-exact* is BSC_{exact} with 4000-instruction chunks. We see that for a few SPLASH-2 applications and for the commercial ones, performance degrades as chunk size increases. Comparing *4000* to *4000-exact*, however, we see that most of the degradation comes from increased aliasing in the signature, rather than from real data sharing between the chunks.

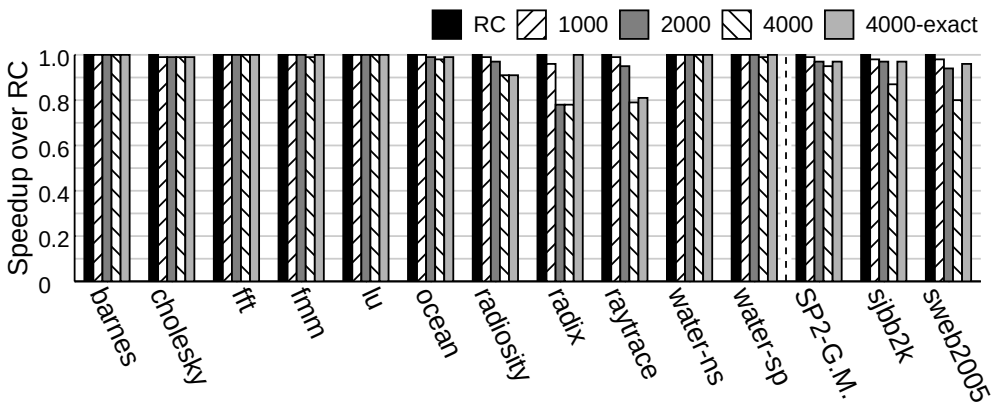


Figure 6.14: BSC_{dypvt} performance with different chunk sizes.

6.6.3 General Characterization of *BulkSC*

Table 6.3 characterizes *BulkSC*. The average concurrency for all benchmarks is very close to 8. Unless otherwise indicated, the data is for BSC_{dypvt}. We start with the *Squashed Instructions* columns. In BSC_{exact}, squashes are due to false and true sharing, while in BSC_{dypvt} and BSC_{base}, squashes also occur due to aliasing in the signatures. While the wasted work in BSC_{base} is 8-10% of the instructions, in BSC_{dypvt} it is only 1-2%, very close to that in BSC_{exact}. This is due to reduced aliasing.

The *Average Set Sizes* columns show the number of cache line addresses encoded in the BSC_{dypvt} signatures. We see that *Priv. Write* has many more addresses than *Write*. The *Priv. Write* data shows that a Private Buffer of ≈24 entries is typically enough. We also note that radix is a peculiar case because chunk set sizes are skewed towards very low and very high values.

The *Spec Line Displacements* columns show how often a line speculatively read or written is displaced from the cache per 100k commits. These events are very rare in SPLASH-2. They are less rare in the commercial applications, where they can be as high as 1 per 10 commits for the read set. Note, however, that in *BulkSC*, displacements of data read *do not* cause squashes, unlike in SC++. The column *Data from Priv. Buff.* shows how frequently the Private Buffer has to provide data. As expected, this number is very low — 6-9 times

Appl.	Squashed Instructions (%)			Average Set Sizes in BSC_{dypvt} (Cache Lines)		
	BSC_{exact}	BSC_{dypvt}	BSC_{base}	Read	Write	Priv. Write
barnes	0.01	0.03	6.27	22.6	0.1	11.9
cholesky	0.04	0.05	2.18	42.0	0.9	11.6
fft	0.01	1.37	2.93	33.4	3.3	22.7
fmm	0.00	0.11	6.99	33.8	0.2	6.2
lu	0.00	0.00	3.29	15.9	0.1	10.8
ocean	0.35	0.92	2.14	45.3	6.7	8.4
radiosity	0.98	1.04	4.25	28.7	0.5	15.2
radix	0.01	10.89	30.75	14.9	5.2	14.4
raytrace	2.71	2.92	8.48	40.2	0.8	12.7
water-ns	0.03	0.07	12.67	20.2	0.1	16.3
water-sp	0.06	0.09	10.23	22.2	0.1	17.0
SP2-AVG	0.38	1.59	8.20	29.0	1.62	13.4
sjbb2k	0.45	1.11	10.33	43.6	3.56	19.2
sweb2005	0.23	0.88	9.97	61.1	3.76	21.5

Appl.	Spec. Line Displacements (Per 100k Commits)		Data from Priv. Buff. (Per 1k Comm.)	# of Extra Cache Invs. (Per 1k Comm.)
	Write Set	Read Set		
barnes	0.3	209.0	0.1	0.1
cholesky	0.0	57.2	1.0	0.2
fft	0.0	0.0	0.1	2.0
fmm	0.0	241.0	0.2	0.5
lu	0.0	0.9	0.0	0.0
ocean	0.0	1206.7	4.9	4.3
radiosity	0.0	50.7	29.9	28.8
radix	0.0	375.6	0.1	1760.0
raytrace	0.0	98.9	30.0	84.3
water-ns	0.0	2.7	0.3	1.9
water-sp	0.0	152.6	0.4	1.4
SP2-AVG	0.0	217.7	6.1	171.2
sjbb2k	1.8	6838.4	6.7	2.9
sweb2005	0.0	10502.5	8.7	4.1

Table 6.3: Characterization of *BulkSC*. Unless otherwise indicated, the data corresponds to BSC_{dypvt}

per 1k commits on average.

When a processor receives the W signature of a committing chunk, it uses it to invalidate lines in its cache. Due to aliasing in the signatures, it may invalidate more lines than necessary. The *Extra Cache Invs.* column quantifies these unnecessary invalidates per 1k commits. With the exception of *radix*, this number is very low and unlikely to affect performance, since these lines are likely to be refetched from L2. *radix* has a very skewed distribution of set sizes, when sets are very large, aliasing increases dramatically.

Appl.	Signature Expansion in Directory			
	Lookups per Commit	Unnecessary Lookups (%)	Unnecessary Updates (%)	Nodes per W Sig.
barnes	0.1	12.7	0.3	0.08
cholesky	1.2	27.7	0.0	0.18
fft	22.1	85.0	0.3	0.01
fmm	0.7	78.0	1.0	0.08
lu	0.1	16.7	0.0	0.01
ocean	9.5	29.9	0.4	0.05
radiosity	0.6	23.2	0.5	1.15
radix	37.8	86.2	0.4	1.10
raytrace	0.8	6.2	0.4	0.95
water-ns	0.2	42.0	0.7	0.74
water-sp	0.0	36.1	4.6	1.12
SP2-AVG	6.7	40.3	0.8	0.50
sjbb2k	4.0	10.1	0.1	0.06

Appl.	Arbiter			R Sig. Required (% Commits)
	# of Pend. W Sigs.	Non-Empty W List (% Time)	Empty W Sig. (% Commits)	
barnes	0.09	8.2	95.3	3.9
cholesky	0.03	2.9	98.1	1.1
fft	0.10	9.4	90.9	1.2
fmm	0.03	3.0	98.2	1.2
lu	0.06	5.7	96.8	2.7
ocean	0.53	40.0	55.8	13.6
radiosity	0.09	8.5	95.2	4.0
radix	0.56	49.3	32.9	15.5
raytrace	0.22	20.6	84.9	8.6
water-ns	0.02	1.4	99.2	0.7
water-sp	0.01	0.5	99.7	0.2
SP2-AVG	0.16	13.6	86.1	4.8
sjbb2k	0.54	46.1	46.9	17.8
sweb2005	0.65	51.7	49.5	28.1

Table 6.4: Characterization of the commit process and coherence operations in BSC_{dypvt}.

6.6.4 Commit and Coherence Operations

Table 6.4 characterizes the commit process and the coherence operations in BSC_{dypvt}. The *Signature Expansion* columns show data on the expansion of W signatures in the directory. During expansion, directory entries are looked up to see if an action is necessary. The *Lookups per Commit* column shows the number of entries looked-up per commit. Since the Write Set sizes are small, this is a small number (7 in SPLASH-2 and 4 in the commercial applications). The *Unnecessary Lookups* column is the fraction of these lookups performed due to aliasing. In SPLASH-2, about 40% of the lookups are unnecessary, while in the commercial applications only 10-17% are. The *Unnecessary Updates* column shows how many directory entries are unnecessarily updated due to aliasing. This number is negligible

(0.1-0.8%). Finally, the *Nodes per W Sig.* column shows how many nodes receive the W signature from the directory. On average, a commit sends W to less than one node — often, the chunk has only written to data that is not cached anywhere else.

The next few columns characterize the arbiter. *Pend. W Sigs.* is the number of W signatures in the arbiter at a time, while *Non-Empty W List* is the percentage of time the arbiter has a non-empty W list. These numbers show that the arbiter is not highly utilized. *Empty W Sig.* shows how many commits have empty W signatures, due to accessing only private data. This number is 86% in SPLASH-2 and 47-50% in the commercial codes. Thanks to them, the arbiter's W list is often empty. Finally, *R Sig. Required* shows the fraction of commits that need the R signature to arbitrate. The resulting low number — 5% for SPLASH-2 and 18-28% for the commercial codes — shows that the *RSig* optimization works very well.

Figure 6.15 shows the interconnection network traffic in bytes, normalized to RC. We show RC (R), BSC$_{exact}$ (E), BSC$_{dypvt}$ without the *RSig* commit bandwidth optimization (N) and BSC$_{dypvt}$ (B). The traffic is due to reads and writes (*Rd/Wr*), R signatures (*RdSig*), W signatures (*WrSig*), invalidations (*Inv*), and other messages.

This figure shows that the total bandwidth overhead of BSC$_{dypvt}$ (B) compared to RC (R) is around 5-13% on average. This is very tolerable. The overhead is due to the transfer of signatures (*WrSig* and *RdSig*) and to extra data fetches after squashes (increase in *Rd/Wr*). The difference between N and B shows the effect of the *RSig* commit bandwidth optimization. We see that it has a significant impact. With this optimization, *RdSig* practically disappears from the B bars. Note that, without this optimization, *RdSig* is very significant in fmm and water-sp. These two applications have few misses and, consequently, signatures account for much of the traffic. Finally, the difference between E and N shows the modest effect of aliasing on traffic.

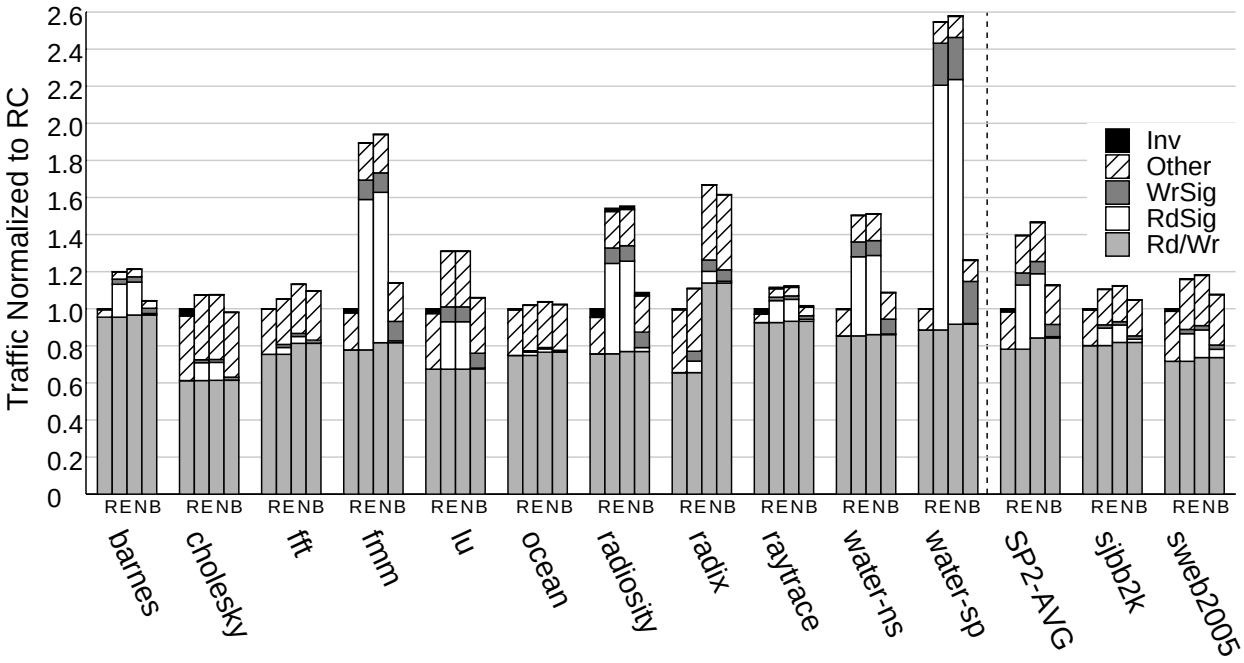


Figure 6.15: Traffic normalized to RC. *R*, *E*, *N*, *B* refer to RC, BSC$_{exact}$, BSC$_{dypvt}$ without the *RSig* optimization, and BSC$_{dypvt}$, respectively.

6.7 Related Work on Sequential Consistency

There is extensive literature on relaxed memory consistency models (e.g., [2, 22, 28]). Discussing it is outside our scope.

Chunks are similar to tasks in TLS or transactions in TM in that they execute speculatively and commit as a unit. In particular, an environment with transactions all the time such as TCC [34] is related to *BulkSC* where processors execute chunks all the time. However, the environments differ in that while tasks and transactions are statically specified in the code, chunks are created dynamically by the hardware. Still, *BulkSC* may be a convenient building block for TM and TLS because it will support SC ordering for the entire program, including transactional with respect to non-transactional code.

The concept of chunk is the same as the concept of *Implicit Transaction* proposed by Vallejo *et al* [72]. In the context of their *Kilo-instruction* multiprocessor project, the authors indicate that sequential consistency can be supported at a coarse-grain by using automati-

cally generated groups of instructions called implicit transactions. We became aware of their paper ([72]) only after publication of our BulkSC work ([16]) in June 2007. However, in their work, they do not present a detailed implementation of the scheme and do not provide an evaluation. In our work, we present a detailed proof of why this approach supports sequential consistency. In addition, we also show a detailed implementation of this approach — in the most challenging environment of a generic network that does not support broadcast. Finally, we perform a detailed evaluation.

Concurrently with our work, Chafi *et al.* [17] have proposed an arbiter-based memory subsystem with directories for TM.

Also concurrently with our work, Wenisch *et al.* [76] have proposed Atomic Sequence Ordering (ASO) and a scalable store buffer design to enable store-wait-free multiprocessors. ASO, like BulkSC, makes a group of memory operations appear atomic to the rest of the system to avoid consistency violations. Our approach in BulkSC is to build the whole memory consistency enforcement based only on coarse-grain operation. In BulkSC, stores are also wait-free because they retire speculatively before the chunk commits.

Part II

Concurrency Control with Data Coloring

Chapter 7

Motivation for Data-Centric Synchronization

Developing a parallel application consists of four steps [51]: decomposing the problem, assigning the work to threads, orchestrating the threads, and mapping them to the machine. After decomposing the problem, orchestration is arguably the most challenging step, as it involves synchronizing the threads. It is in this area that innovations to simplify parallel programming are most urgently sought.

One such innovation is Transactional Memory (TM) [5, 34, 37, 54, 60]. In TM, the programmer specifies sequences of operations that should be executed atomically. TM simplifies parallel programming in two ways. First, the programmer does not need to worry about the intricacies of managing locks. Second, he does not need to fine-tune critical sections as much, since concurrency is only limited by dependences — not critical section length.

We claim, however, that TM is still complicated: it requires the programmer to reason *non-locally*. Specifically, when the programmer inserts a transaction annotation, he also needs to think about what other parts of the program may be accessing this same or related shared data, and potentially insert transaction annotations there as well. Intuitively, like inserting lock and unlock operations, inserting transaction annotations involves taking a *code-centric* approach.

To improve programmability further, we need a *data-centric* approach [73]. With *Data-Centric Synchronization* (DCS), the programmer associates synchronization constraints with the program's data structures. Such constraints indicate which sets of data structures should remain consistent with each other and, therefore, be accessed in the same critical section. From these constraints, the system automatically infers the critical sections and inserts

thread synchronization operations in the code. DCS simplifies parallel programming because the programmer reasons *locally*, focusing only on what structures should be consistent with each other.

Existing DCS proposals [73] take user-provided, data-centric synchronization constraints and decide where to insert critical sections using software-only support. In particular, the compiler needs to analyze all the accesses in the code. This is unrealistic in most C/C++ environments, where pointer aliasing is common and, most importantly, dynamic linking denies the compiler access to the whole program.

To make DCS practical, this thesis proposes the first design for Hardware DCS (H-DCS). Our proposal, called *Colorama*, relies on two hardware primitives: one that monitors all memory accesses to decide when to start a critical section, and one that flexibly triggers the exit of a critical section. Colorama is independent of the underlying synchronization mechanism. In this thesis, we present a transaction-based implementation and also discuss the issues that appear in a lock-based implementation.

We describe Colorama's architecture, a simple implementation that extends a Mondrian Memory Protection (MMP) [77] system, its programming model and API, and its capacity to help debug conventional codes. We show that Colorama needs few hardware resources and has small overhead. It supports general-purpose, pointer-based languages such as C/C++ and, in our opinion, can substantially simplify the task of writing new parallel programs.

Part II is organized as follows: Chapter 8 introduces DCS; Chapter 9 describes Colorama's architecture and its implementation issues; Chapter 10 discusses the programming environment and debugging issues; Chapter 11 shows an evaluation of the main ideas and architectural components; and Chapter 12 discusses related work.

Chapter 8

Basic Idea of Data-Centric Synchronization

In Data-Centric Synchronization (DCS) [73], the programmer associates synchronization constraints with data structures — typically when they are declared or allocated. These constraints specify which data structures are in the same "data consistency domain" and, therefore, should be kept consistent with each other. This means that when one structure is being modified, all the other structures in the same domain need to be protected from access by other threads. To support this model, when a thread accesses a structure of a domain, the thread automatically enters a critical section for that domain. No other thread can now access structures of that domain. When the thread finishes working on structures of that domain, the thread automatically exits the critical section.

DCS is in contrast to conventional Code-Centric Synchronization (CCS), where synchronization constraints are associated with code. In CCS, the programmer marks what code is inside which critical section.

We argue that DCS has a significant advantage over CCS in *programmability*. CCS requires the programmer to reason *non-locally* [73]: every time he inserts a transaction begin/end or a lock acquire/release annotation in the code, he also needs to think about what other locations in the program may be accessing this same or related data structures, and potentially insert synchronization annotations there as well. Instead, with DCS, the programmer reasons *locally*, focusing only on what data structures should be consistent with each other. The system automatically infers the critical sections.

The shortcoming of DCS stems from limited program knowledge. The system has to automatically infer when the code enters and exits a critical section, so that it can insert

the appropriate synchronization operations around the section.

Identifying entry points to critical sections largely involves identifying accesses to data structures belonging to a domain. Identifying exit points is harder. It is typically impossible for the system to know when a thread has stopped working on structures of a given domain and, therefore, the critical section for that domain should terminate. Consequently, DCS schemes have an *Exit Policy*, which is a simple, clear algorithm for terminating a critical section. The exit policy used by the system is communicated to the programmer. This is because, to write correct code, the programmer *needs to know* the exit policy used, and write code in agreement with it. We believe that having a simple exit policy is an acceptable burden given the improvement in programmability provided by DCS.

8.1 Software DCS (S-DCS)

DCS has only been implemented in software, under limited environments. The main example of what we strictly consider Software DCS (S-DCS) is Vaziri *et al.*'s Atomic Sets [73]. This system includes a compiler and language extensions to Java. The programmer, when declaring Java classes, can group several fields into an Atomic Set. The elements of an Atomic Set are supposed to be manipulated atomically inside critical sections that are automatically created by the compiler.

The entry points of critical sections of an Atomic Set are inferred by the compiler by statically analyzing the code and identifying likely accesses to data belonging to the Set. Since Java is relatively analyzable due to type safety and the lack of pointer arithmetic, if the compiler has access to the whole program, then it can conservatively identify when data from Atomic Sets are accessed [73].

The exit policy used by Vaziri *et al.* is to insert the exit point of a critical section right before the return of the Java method that contains the corresponding entry point. This policy builds on the intuition that a method is a natural unit of work — a method is

typically exited when the work is completed. Therefore, a single method includes both the entry and the exit points of a critical section.

8.2 Proposal for Hardware DCS (H-DCS): *Colorama*

S-DCS is unsuitable for popular languages such as C/C++, which allow pointer arithmetic and aliasing. Since the compiler cannot fully analyze the code due to lack of pointer information, it can only generate conservative critical section approximations of very limited use. Alternatively, if it inserts instructions to check the address of every pointer access dynamically, it induces intolerable overhead. In addition, in environments with dynamic linking, deployment of S-DCS is impractical because the compiler may lack access to the whole program.

Therefore, this thesis proposes a novel architecture to support DCS in hardware. The resulting *Hardware DCS (H-DCS)* scheme is called *Colorama*. It supports any type of access pattern, has low overhead, and is usable in any language.

Colorama has two primitives, corresponding to the need to identify critical section entry and exit points. The first one is hardware to monitor all addresses issued by the processor with very low overhead. If a thread accesses a structure belonging to a consistency domain from outside of a critical section for that domain, Colorama starts a critical section.

The second primitive is hardware to support the exit of a critical section. Such primitive is very flexible and is driven by the compiler, so that different exit policies can be supported. At all times, however, it has to be clear to the programmer what exit policy will be used by the compiler as it generates the executable. As a starting point, in this thesis we use the exit policy used by Vaziri *et al.* [73] because it is very intuitive. For example, Wang and Stoller [75] use the heuristic that methods execute atomically to identify potential atomicity violations in Java programs.

Note that the support for Colorama does not replicate (and is largely independent of) the

support that the machine provides for synchronization. In this thesis, we propose a Colorama implementation that relies on transactions as the underlying synchronization mechanism. We also discuss the issues that appear in an implementation based on locks.

8.3 Examples of Colorama Programming

In Colorama, a data consistency domain is called a *Color*, while a memory region with structures belonging to a consistency domain is referred to as *Colored*. In this section, we show three motivating examples.

Linked List. Consider a linked list that is manipulated by functions that insert a node, delete a node, and traverse the list (Figure 8.1). The programmer can color all the nodes in the list with the same color. This is done with the *color* and *colorprop* system calls shown. *Color* takes a starting address, a size, and a color ID; it colors the address range with color ID. *Colorprop* takes a starting address, a size, and a colored address; it propagates the color of the colored address to the address range.

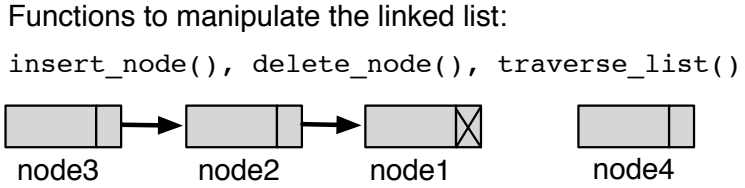


Figure 8.1: Example of linked-list manipulation.

With Colorama's support, the list manipulation functions in the figure are written without any transaction or lock annotation. The result is code as simple as in a sequential program.

Task Queue. Consider a task queue where each entry points to a bucket of shared data

(Figure 8.2). A thread accesses the task queue to retrieve a bucket. Then, the thread operates on the bucket. Finally, it accesses the task queue again to deposit new buckets. There are several variables associated with the task queue: head and tail pointers, a flag to check if the queue is empty, and a count of threads waiting on an empty task queue. The programmer can color the task queue, head, tail, empty and *num_waiters* structures with a single color, and each of the data buckets with a different color. Then, all the functions listed in the figure are written with no transaction or lock annotation.

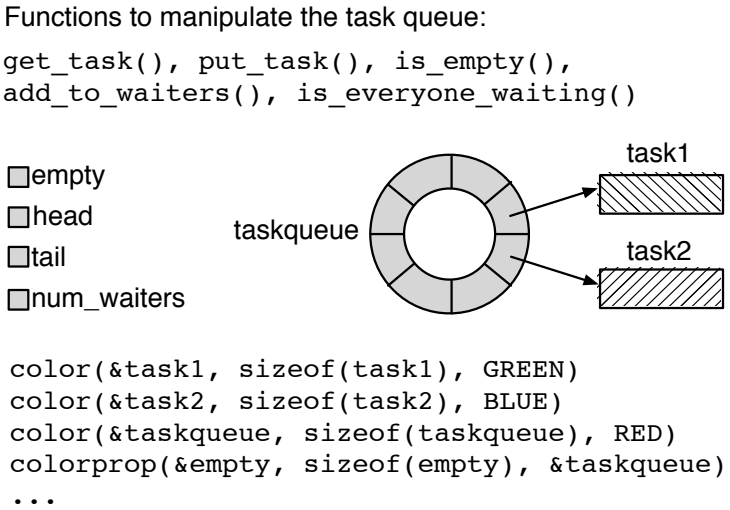
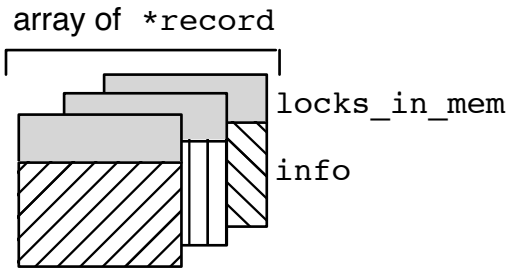


Figure 8.2: Example of task queue handling.

Sample MySQL Structure. Figure 8.3 shows a data structure from the MySQL database that is composed of many records. Each record has the *locks_in_mem* field and the *info* set of fields. A single global lock protects the *locks_in_mem* field in all records. Such lock is accessed from 29 sites in the MySQL code. Each record's *info* is protected by a per-record lock. Such lock is accessed from 14 sites. A Colorama programmer can color *locks_in_mem* in all records with the same color, and the per-record *info* fields with a per-record color. The records can now be accessed with no transaction or lock annotation.




```
for(i=0; i < MAXREC, i++) {
  color(&record[i]->locks_in_mem, ptrsize, RED)
  color(&record[i]->info, infosize, RED+i+1)
}
```

Figure 8.3: Sample structure from MySQL.

Chapter 9

An Architecture for Data-Centric Synchronization

9.1 Overview

Colorama's architecture consists of a structure shared by all threads and a per-thread structure. The shared structure contains the current list of colored regions, while the per-thread one specifies what colors are currently owned by the thread. The per-thread structure also includes the mechanism to support the exit of a critical section.

At every load and store, Colorama leverages efficient hardware (Section 9.6) to check with very low overhead whether both the address is colored and the thread does not own the color. If so, Colorama triggers the entry to the color's critical section. Later, when certain events specified by the exit policy are detected, Colorama triggers the exit from the color's critical section.

The shared structure is called Color Map, or *Palette* (Figure 9.1). It is a software structure in shared memory that is partially cached in special hardware at each processor. The Palette lists, for each currently colored address region, the start and end addresses and its color (*ColorID*). Multiple address regions — and therefore multiple Palette entries — can have the same ColorID. However, a given address can only have a single ColorID and, therefore, appear in a single entry.

The per-thread structure is the *Thread Color Status*. It contains the set of ColorIDs currently owned by the thread. These are the colors whose critical sections are currently being executed by the thread. They are listed in the Owned Colors Array.

The Thread Color Status also provides an efficient hardware primitive for the software to

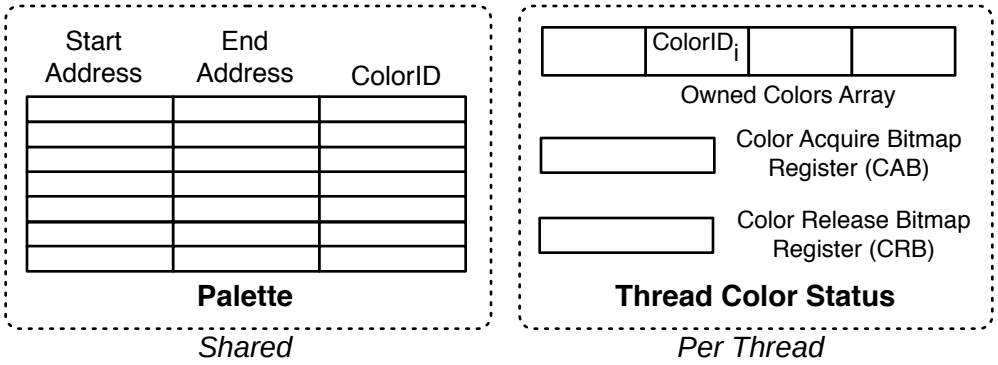


Figure 9.1: Architectural support for Colorama. While the Palette is conceptually a table, it has a hardware-software distributed implementation (Section 9.6.1).

implement the exit policy. The primitive is built around the two Color Bitmap Registers: the read/write Color Acquire Bitmap (CAB) register and the write-only Color Release Bitmap (CRB) register (Figure 9.1). These registers have as many bits as entries in the Owned Colors Array (e.g., 64). Every time that a ColorID is inserted in location i of the Owned Colors Array, the corresponding bit in the CAB register is automatically set in hardware. In addition, when the software sets bit i of the CRB register, the hardware triggers a critical section exit for the ColorID in the corresponding entry of the Owned Colors Array.

9.2 Chosen Critical Section Exit Policy

As indicated in Section 8.2, in this thesis we choose the exit policy used by Vaziri *et al.* [73]: trigger the exit of a color's critical section when the thread returns from the subroutine where the critical section was entered. We choose it because it is simple and intuitive: a subroutine is a natural unit of work; when the subroutine returns, the thread is likely to have finished the operation it was doing and, therefore, stopped working on that color's structures. Some evidence that programmers already follow this convention informally is presented later (Section 11.3.1). Note, however, that in DCS, writing correct code *requires* that the programmer be aware of the exit policy supported by the system and follows it.

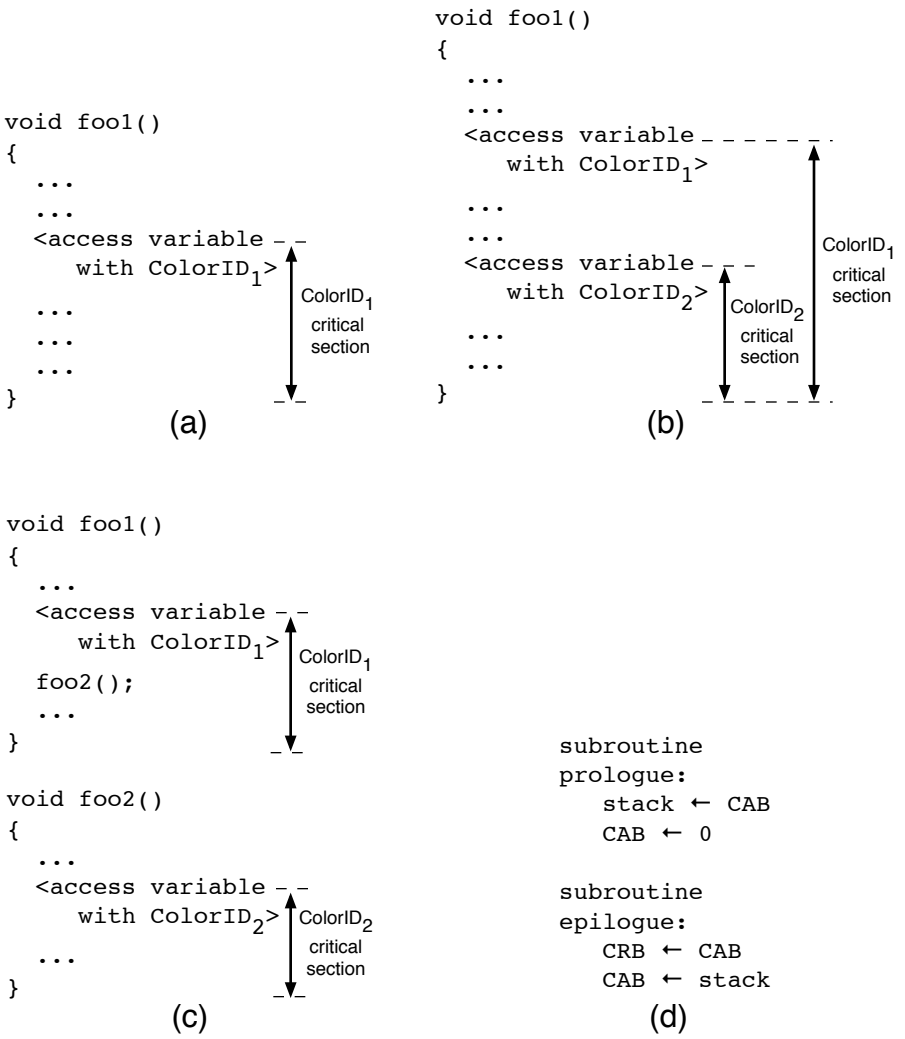


Figure 9.2: Illustration of the policy chosen in this thesis to exit critical sections in Colorama and its implementation.

Abstractly, what this exit policy achieves is to make all memory operations on colored data inside a method appear to execute atomically.

Figure 9.2 illustrates the policy. The figure assumes that variables A and B are colored with $ColorID_A$ and $ColorID_B$, respectively. Figure 9.2(a) shows an access to A, and how the resulting critical section runs until the end of the subroutine. Figures 9.2(b) and 9.2(c) show how critical sections nest. In both cases, a thread accesses A and, before it returns from the subroutine, it accesses B. As a result, the $ColorID_B$ critical section is nested inside the $ColorID_A$ one. The two figures, however, show different cases. In Figure 9.2(b), the

accesses to A and B are in the same subroutine; as a result, both critical sections finish at the same time. In Figure 9.2(c), the accesses to A and B are in different subroutines, and the sections finish at different times.

This policy is implemented with the *compiler-inserted* instructions shown in Figure 9.2(d). At every subroutine entry, the compiler saves the CAB register in the stack and then clears it. This does not affect the Owned Colors Array (Figure 9.1). As the subroutine executes, if a *new* color becomes owned, the corresponding bit in the CAB register gets automatically set. Before the subroutine returns, the compiler copies the CAB to the CRB register, thereby triggering the exit of all the critical sections entered in this subroutine. Then, it restores the CAB register from the stack, leaving it in the state it had before the subroutine was called. This algorithm works with any nesting.

Another possible exit policy is to have the programmer specify points in the code where the colored data is supposed to be consistent. This is equivalent to explicit color releases. This way, when such markers in the code tell that a color is consistent, if the color is held by the current thread, the exit of the corresponding critical section could be triggered. This thesis does not elaborate on such policy, however it is possible to implement it just using the API described 10.3.

9.3 Detailed Colorama Operation

Based on the previous discussion, we now describe the operation of Colorama in detail. At every load and store, the cached Palette and the Thread Color Status are checked in hardware. If the address belongs to a colored region and the thread does not own that ColorID, a Colorama user-level software handler is automatically invoked with low overhead.

The handler adds ColorID to the Owned Colors Array. Then, if nested transactions are supported, the handler starts a new transaction for that color; if only flat transactions are supported, it starts a new transaction only if this is the only color owned by the thread.

The handler then returns to the program. While these simple operations could be done in hardware, using a software handler is more flexible.

As per our exit policy, before every subroutine return, an instruction stores to the CRB register. For each set bit that gets written to the CRB register, if the same-offset entry in the Owned Colors Array has a valid ColorID, the hardware triggers a critical section exit for that ColorID.

A section exit for a set of ColorIDs starts with the automatic invocation of a Colorama user-level software handler. For each ColorID, the handler performs the following operations. First, the handler removes that ColorID from the Owned Colors Array. Then, if this was the last color in the structure, the handler initiates a transaction commit. If this was not the last color and the machine supports nested transactions, the handler initiates an inner-transaction commit for that ColorID. What an inner-transaction commit does is independent of Colorama. It could, for example, create a new checkpoint while keeping the thread speculative, in order to minimize the rollback distance in case of a collision. Finally, the handler returns.

When a transaction is squashed, its ColorID(s) are removed from the Owned Colors Array and its bit(s) in the CAB register are cleared.

9.4 Pointers as Subroutine Arguments

Sometimes, a critical section performs multiple operations on a structure, and invokes one subroutine per operation — passing as argument to each subroutine a pointer to the structure. This is common when handling complex structures such as hash tables. Figure 9.3(a) shows a lock-based example of a read and a write to a hash table. *htPtr* is a pointer to the hash table.

Figure 9.3(b) shows the corresponding Colorama code, where we assume that the hash table is colored. Colorama's hardware will detect accesses to the hash table only inside

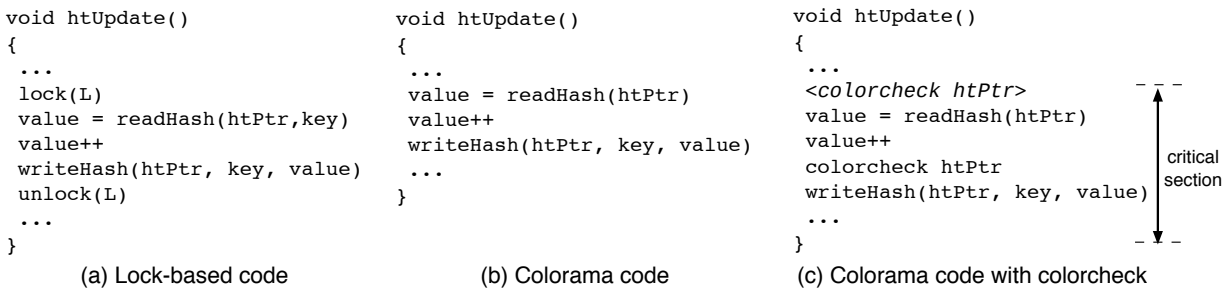


Figure 9.3: Using the colorcheck instruction.

subroutines *readHash()* and *writeHash()*. As a result, it will create two separate critical sections, one inside each subroutine. This is not what the programmer intended.

Since we believe that this is a common style of programming, we would like Colorama to enclose the two subroutines inside a single critical section. Interestingly, Colorama would automatically do so if we accessed the hash table in subroutine *htUpdate()* before the call to *readHash()*: the exit policy would extend the critical section from that point till the end of *htUpdate()*.

To support this case, we extend Colorama with a primitive to potentially start a critical section. The mechanism is a new *colorcheck* instruction that performs a run-time address check. Colorcheck takes an address and checks whether it is colored and the color is not owned by the thread. If so, Colorama automatically triggers a critical section entry as usual (Section 9.3). Colorcheck does not read or write the address, and cannot raise protection exceptions.

To use this primitive for our purposes, we extend the Colorama compiler to identify subroutine calls with arguments that are pointers. For every such argument, the compiler inserts a colorcheck instruction with that argument, right before the call — in the example, the argument is *htPtr*. The resulting code is shown in Figure 9.3(c). This change accomplishes what we need. At run time, colorcheck checks the contents of *htPtr* before *readHash()* and triggers the start of the critical section.

9.5 Why Use Multiple Colors

If the system supports nested transactions, having multiple colors provides an intuitive way to build transaction nests [56]: every time a new color is accessed inside a transaction, a new nesting level is created.

Irrespective of whether or not the system supports nested transactions, having multiple colors is also useful in three ways. First, it can help debug the code. Specifically, every time a processor attempts to commit a transaction, as it broadcasts the addresses that it wrote, we propose that it also broadcast the colors that the transaction owned. If a second processor that is executing a different-color transaction detects a collision with the committing one, the programmer is warned that a bug is likely — different-color transactions should not have collisions. If they do have a collision, this indicates that the programmer was not aware of some shared data, because the conflict must have been on uncolored data — which is supposed to be private.

The second use is to help optimize the cross-thread dependence disambiguation that takes place at thread commit. If we are certain that the code has no bugs, we may decide to reduce overheads by not checking for collisions between concurrent transactions of different colors. This may save inter-processor traffic.

The final advantage of supporting multiple colors is that it enables the programmer to embed more information in the program on how shared data are used.

If the system uses locks, instead, supporting multiple colors directly translates into enabling more concurrency (Section 9.6.3).

9.6 Implementation Issues

9.6.1 Colorama Structures

The Colorama structures are the Palette and the Thread Color Status (Figure 9.1). The Palette is a distributed structure implemented part in hardware and part in software. It is accessed with a pattern similar to that of structures that contain address protection information — i.e., which address can be read or written by which thread. Indeed, protection information is also shared by all threads and is accessed at every memory request. Consequently, both types of information can share the same implementation. One difference is that the Palette contains per-word information, while current virtual memory systems associate protection information with pages. Consequently, to accommodate the Palette, we would need to redesign current TLB structures. In practice, there is already an efficient design that manages per-word protection information, namely the Mondrian Memory Protection (MMP) system [77]. Therefore, we implement the Palette as extra bits to be stored in the MMP structures.

The implementation of an MMP system is shown as the white structures of Figure 9.4(a). The Multilevel Permissions Table is a software table in shared memory that holds all the protection information. The table is hierarchically organized for space efficiency, with ranges of addresses expanded enough to keep the protection information at the available grain size (word, page, etc.). Processors transparently cache on demand sections of the table in a hardware buffer called Protection Lookaside Buffer (PLB). In addition, for faster access to protection information, architectural registers have sidecar registers, with recently-accessed protection information. Loads and stores automatically access the sidecars and PLB in hardware to check permissions. A PLB miss is like a TLB miss, and brings in the permissions transparently. OS-initiated PLB/sidecar updates propagate to memory and invalidate relevant entries in other processors' PLBs and sidecars.

The shaded fields in Figure 9.4(a) constitute the Palette. They simply add the ColorID

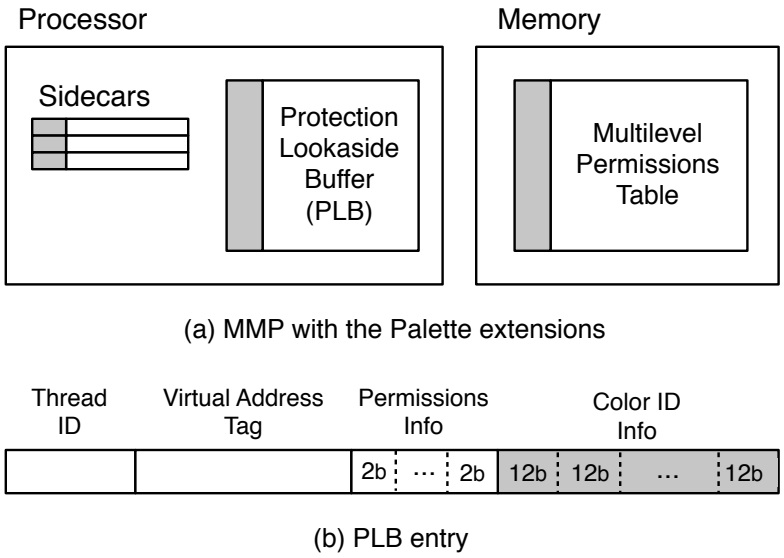


Figure 9.4: Implementation of the Palette on top of an MMP system. The shaded fields constitute the Palette.

bits to the three MMP structures. Figure 9.4(b) shows a PLB entry in detail. A PLB entry may correspond to a cache line. The Palette adds a ColorID (e.g., 12 bits) to every word contained in the PLB entry — e.g., 16 × 12 bits for a 16-word line. A load or store automatically checks the ColorID of the address accessed, which is typically in a sidecar register or in the PLB. When a thread changes the color of a range of addresses, the OS updates the PLB and the other structures as in the MMP system.

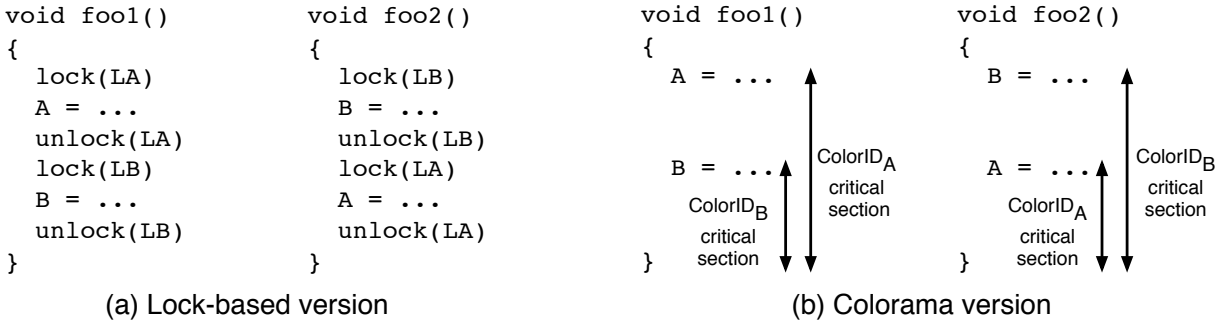


Figure 9.5: Example of how the chosen exit policy may cause a deadlock. Implementations with transactions do not have this problem.

The Thread Color Status consists of three structures accessible in user mode: the

read/write Owned Colors Array, the read/write CAB register, and the write-only CRB register (Figure 9.1). They hold and manage the colors owned by the currently-running thread. These three structures all have the same number of entries (e.g., 64), although each entry is one bit in the registers and a ColorID in the Owned Colors Array. The Owned Colors Array and the CAB register are saved on a context switch. If a thread temporarily needs to own more colors than entries available, Colorama traps to software, which manages the extra state required.

The other key Colorama features are the colorcheck instruction, a related instruction called *getcolorid* (whose purpose is discussed later), and the low-overhead invocation of user-level handlers. The colorcheck and getcolorid instructions take an address. They are implemented like a load, in that the hardware accesses the sidecar, PLB entry, or the Permissions Table entry for the address. The getcolorid instruction simply returns the ColorID (if any) of the address. The colorcheck instruction, again like a load, if it finds that the address is colored and that the ColorID is not in the Owned Colors Array, it triggers a critical section entry. However, unlike a load, colorcheck stops right there, and does not access memory. Neither colorcheck nor getcolorid raises protection exceptions.

When a thread needs to enter or exit a critical section, the hardware invokes a Colorama user-level software handler. Using a software handler adds flexibility and simplicity, but it must be triggered with low overhead. Fortunately, the handler does not require any change in privilege mode. We can use support such as that of Informing Memory Operations [40].

The maximum number of colors supported is hardwired in several structures. While most programs need about 1K or fewer colors (Section 11.3.3), we size Colorama for a large number (4K). If the program needs more colors, Colorama hashes multiple colors into one. In this case, performance may be affected. Specifically, given the uses of multiple colors in a system that uses transactions (Section 9.5), we may end up combining two transactions that should be nested (and therefore squashing more work than necessary on a collision), potentially missing bug warnings, or generating more traffic than necessary to check for

collisions.

9.6.2 Coloring at Page Granularity

An alternative implementation involves restricting color assignment such that all the structures in the same page share the same color. This policy can be enforced by specifying colors at memory allocation time and extending the memory allocator algorithm to keep pools of colored memory.

Such approach would need a simpler Palette implementation, since we could extend TLB and page table entries to include color information — the MMP system would not be needed. However, the resulting coloring support would be less flexible and possibly more complex for the software, since the color of the data structures would affect the memory layout.

9.6.3 Using Locks as the Underlying Synchronization Mechanism

This thesis proposes an implementation of Colorama on a machine that uses transactions as the underlying synchronization mechanism. It is also possible to build Colorama on a system that uses locks. In this case, each distinct color is associated with a different implicit lock.

The Colorama user-level handler invoked at the entry point of a critical section, instead of starting a transaction, attempts to acquire the lock corresponding to the color. When it succeeds, it adds the ColorID to the Owned Colors Array and returns. Similarly, the handler invoked at the exit of a critical section releases the corresponding lock, removes the ColorID from the Owned Colors Array and returns. Note also that it is not possible to hash multiple colors into one because deadlocks may happen.

In a lock-based implementation, the specific exit policy that we have chosen in this thesis may have two effects. The first one is that, since critical sections now run until the end of subroutines, they tend to have larger sizes and, therefore, may cause an increase in

lock contention. In practice, we show in Section 11.3.2 that the average increase in critical section size is likely to be modest.

The second effect is that, depending on how the code is written, the exit policy chosen may cause deadlocks. As an example, Figure 9.5(a) shows a lock-based code and Figure 9.5(b) shows its corresponding Colorama code. In Figure 9.5(a), *foo1* acquires and releases lock LA and then acquires and releases lock LB, while *foo2* performs the same operations in opposite order. Suppose that, under Colorama, variables A and B have colors $ColorID_A$ and $ColorID_B$, respectively. Because of the exit policy, *foo1* will nest $ColorID_B$'s critical section inside $ColorID_A$'s, and *foo2* will do the opposite. If two threads executing *foo1* and *foo2*, respectively, perform the first assignment in *foo1* and *foo2* at the same time, they will deadlock.

This scenario must be rare in practice, since our experiments of Section 11.3.2 on conventional code have been unable to detect even a single instance of subroutine pairs that *could* deadlock in Colorama. Consequently, it may be acceptable to use this exit policy and, rather than trying to avoid deadlocks, detect them and break them if they occur. Alternatively, we can use a different exit policy that is not subject to this problem. We are currently working on this issue.

Deadlocks can be detected with a software table in memory that lists, for each color, the current owner thread and the spinning threads. When the Colorama user-level handler that attempts to acquire the lock for a color fails to do so, it registers its thread ID as spinning on the lock. It then checks for a cycle in owner and spinning thread IDs across multiple locks in the table. If it finds one, a deadlock has occurred. Then, the handler informs the user of where the deadlock happened.

We consider this support to be a debugging aid. We expect that, as programmers become familiar with Colorama's programming model and whatever exit policy is used, they will write code that executes fast and reliably.

Note that deadlocks do not exist in a transaction-based implementation of Colorama.

Transactions are known to be susceptible to livelocks, but they are easily avoided.

Chapter 10

Programming with Colorama

The goal of Colorama is to simplify parallel programming. One of the ways in which Transactional Memory (TM) simplifies the programmer's job is by not requiring so much fine-tuning of the critical sections — concurrency is limited by dependences, not critical section length. With Colorama, the programmer's job is further simplified beyond TM because he does not even need to mark critical sections — the system automatically infers them. Potentially, the result is highly programmable and maintainable code. In this section, we examine several programming issues in Colorama.

10.1 Correctness

At a minimum, Colorama guarantees that all executions of critical sections of the same color by different threads are serializable. Consequently, if the programmer colors all the shared data structures that should be accessed in an exclusive manner, Colorama produces a data-race free program. All conflicting accesses will be separated by transaction boundaries or lock operations.

The extent and granularity of coloring typically matter relatively little in a transaction-based implementation of Colorama, since concurrency is only limited by data dependences — although long transactions with resulting cache overflow are slow. However, they matter substantially more in a lock-based implementation. In this case, if the programmer colors structures for which the accesses do not need to be constrained (e.g., thread-private variables), the resulting superfluous critical sections or longer-than-necessary ones may limit

concurrency and lower performance. Conversely, a programmer can enable more concurrency if variables that do not have mutual consistency constraints are assigned different colors. This may improve performance.

If the programmer fails to color a structure that should be accessed in an exclusive manner, the program may have data races. Likewise, if he assigns different colors to structures that have mutual consistency constraints, or if he does not respect the exit policy of the system — in our case, by continuing to manipulate an exclusive structure past the corresponding subroutine return — the program may function incorrectly.

10.2 Code Compatibility Issues

A program written for Colorama may be linked with libraries that do not use Colorama's Application Binary Interface (ABI) — for example, they use explicit transactions or locks. In this case, no special action needs to be taken. The legacy library will use transactions or locks to protect its own data structures, not program data. For library-accessed program data, Colorama will continue to trigger critical section entries on access and (if the library executes program code through a callback) critical section exits on subroutine returns.

In certain exceptional cases, applications may require the absence of Colorama's default exit policy. For example, consider an infinite loop where a consumer thread reads data from a shared buffer that is filled by a producer thread. If programmed with transactions, every access to the buffer would be a transaction. In Colorama, if the shared buffer is colored, the whole infinite loop would become a single critical section. To avoid this case, the programmer (or compiler) has to explicitly release the buffer's color at every iteration. As another example, to implement a wait on condition variables, the programmer (or compiler) will want to be able to temporarily release a color and then re-acquire it.

These operations are available through a Colorama library as follows. First, consider releasing the color associated with an address. The library first uses a Colorama instruction

called *getcolorid* (Section 9.6.1). Such instruction simply returns the ColorID of the address. Then, the library searches the Owned Colors Array (Figure 9.1) to find the array offset where that ColorID is stored. If found, the library writes to the CRB register a set bit at the same offset, which triggers the release of ColorID. Note also that we can release all colors by writing all ones to the CRB register.

Releasing a color *temporarily* involves releasing the color as before and saving the address. Re-acquiring a color involves using the colorcheck instruction on the saved address.

10.3 Colorama's Complete API

Colorama's complete API is shown in Table 10.1. It contains five instructions, three system calls, and four library calls. The instructions are colorcheck, getcolorid, and moves to/from CAB or CRB. The system calls color or decolor addresses. The reason why these operations are system calls is that they update the PLB, which also contains protection information (Section 9.6.1). These system calls are typically issued when data structures are allocated or deallocated — they are rarely issued otherwise. Possibly, the two coloring system calls could be inserted directly by the compiler, based on language syntax extensions that specify colors when data structures are declared. Moreover, the decolor system call could be inside *free()*. Finally, the rationale for the four library calls in Table 10.1 was presented in Section 10.2. Typically, only experienced programmers would use the library calls.

10.4 Example: Prevention of an Atomicity Violation

Finally, to showcase the advantages of Colorama's programming simplicity, we show one example where Colorama helps prevent a subtle synchronization defect. Figure 10.1 shows Java method *append*, which appends one string to another. It calls methods *length* to get the length of a string and *getChars* to copy the string. The figure also shows a call to append

Instructions (Typically inserted by the compiler)	
`colorcheck Addr`	Check if (Addr is colored and its color is not owned by the thread). If true, enter critical section
`getcolorid Addr, reg`	Save the ColorID of Addr in a register
`mov reg, CAB`	Update the CAB register
`mov CAB, reg`	Read the CAB register
`mov reg, CRB`	Update the CRB register
System Calls (They change the Palette. Inserted by the programmer or the compiler)	
`color (StartAddr, Size, ColorID)`	Color this address range with ColorID
`colorprop(StartAddr,Size,ColoredAddr)`	Propagate the color of ColoredAddr to this address range
`decolor (Addr)`	Remove the color from the structure at Addr
Library Calls (They change the Thread Color Status. Used in exceptional circumstances)	
`color_release ()`	Thread releases ownership of all its colors
`color_release (Addr)`	Thread releases ownership of the color of the structure at Addr
`color_temp_release (Addr)`	Thread *temporarily* releases ownership of the color of the structure at Addr
`color_reacquire ()`	Thread re-acquires ownership of all the colors that it temporarily released

Table 10.1: Colorama's complete API.

string *sb* to string *sa*.

Method *append* is annotated as *synchronized*, which means that it executes under mutual exclusion with other *synchronized* methods invoked on *sa*. Methods *length* and *getChars* are also *synchronized*. However, when they are called from within *append* in the example, they are *synchronized* with other methods invoked on *sb*. As a result, although the individual interactions of *length* and *getChars* on *sb* are atomic, the sequence of interactions is not: it can happen that string *sb* is altered by another thread in-between the *length* and *getChars* calls — resulting in a stale value of *len* at the point of calling *getChars*.

In Colorama, defects such as this one are prevented. If string *sb* is colored, as soon as it is first accessed inside *append*, a critical section starts. With the exit policy used, the critical

```
class StringBuffer {
   public synchronized StringBuffer append(StringBuffer sb)
   {
     ...
     int len = sb.length();
     ...
     sb.getChars(len,...); // len may be stale
     ...
   }
   public synchronized int length() { ... }
   public synchronized void getChars(...) { ... }
}

StringBuffer sa;
StringBuffer sb;
...
sa.append(sb);
```

Figure 10.1: Example where Colorama prevents an atomicity violation.

section extends to the end of the method — therefore encompassing the calls to *length* and *getChars* and avoiding the problem. No code annotations are necessary beyond coloring. Also, note that, if *sb* is not shared, we avoid any synchronization overhead by simply not coloring it.

10.5 Code Debugging Issues

While we argue that programming in Colorama is simpler and less error-prone than in the conventional CCS approach, it is still possible to have bugs. In this section, we examine how to debug Colorama code. In addition, we also consider a related question, namely leveraging the Colorama hardware to debug conventional CCS code.

10.5.1 Debugging Colorama Code

We classify Colorama bugs into three classes: (i) failing to color a structure that should be colored; (ii) coloring two structures from the same consistency domain with two different

colors; and (iii) violating the exit policy. The bugs in class (i) can lead to data races, which can be detected with conventional data-race detection tools. They can also lead to collisions between critical sections of different colors, which are easily detected by Colorama (Section 9.5).

The bugs in classes (ii) and (iii) cause atomicity violations. They can be debugged with conventional tools that use heuristics to detect atomicity violations [23, 75].

The bugs in class (iii) are unique to DCS. For the exit policy used in this thesis, they occur when the programmer assumes that a critical section extends past its corresponding subroutine return. The exit policy, of course, triggers a critical section exit at that particular return. Fortunately, we can use simple heuristics to identify possible instances of these bugs. The procedure is to record the colors of the critical sections that exit at a given subroutine return *i*. Then, we check if the thread accesses any of these colors again before the next N dynamic subroutine returns — where N can be 1. If it does, the programmer is warned, as he may have expected that the color's critical section had extended beyond the return *i*. Note that this procedure only relies on single-thread information — not on information dependent on the access interleaving of multiple threads. As a result, the bug manifests deterministically.

10.5.2 Debugging CCS Code with Colorama Hardware

A programmer who writes conventional CCS code on a machine with Colorama hardware can benefit from additionally annotating the data structures with colors as in DCS. Such annotations, if they drive the Colorama hardware without actually starting critical sections, can help debug the CCS code. As an illustration, assume that the programmer has written the CCS code with transactions. In this case, the Colorama hardware can detect when the following rules are violated, which is a strong indication of a bug.

1. Colored data should only be accessed inside transactions. Accesses from outside are typically bugs.

2. As indicated in Section 9.5, transactions of different colors should not collide. The Colorama hardware records the colors accessed by each transaction. A collision between two transactions of different colors likely suggests that the programmer was unaware of some data sharing.

3. A non-nested transaction should typically access only one color. If a transaction accesses multiple colors, there may be an opportunity for transaction nesting that could be flagged to the programmer. More than a bug, this is possibly a missed optimization opportunity.

4. A subroutine should not typically contain two transactions of the same color. As pointed out in [75], functions that manipulate shared data in parallel programs are often intended to be atomic. Therefore, having two transactions of the same color in the same subroutine rather than one may be a bug.

Chapter 11

Evaluation of Architecture Support for Data-Centric Synchronization

11.1 Quantitative Justification

We support our claim that data-centric concurrency control is the common case by analyzing synchronization patterns in three large software packages: MySQL, WebSphere and Sun's Java Runtime Environment (JRE). Table 11.1 summarizes the collected data.

MySQL is a large parallel application with thousands of files and over a million lines of code, written in C. Its critical sections operate under the protection of thousands of different lock instances. We inspected all critical sections in the code and classified them as data-centric or operation-centric. Figure 11.1 shows examples of two such critical sections. Most critical sections (84%) in MySQL are data-centric. Among the many locks used, Table 11.2 shows three global locks with a large number of critical sections spread over many files. In this code-centric, non-local reasoning using critical sections for large applications, it is easy to overlook the need for a critical section and introduce data races. Using a data-centric model, the programmer simply colors the data-structures and marks the places in the code where data is consistent. All accesses to shared data would be guaranteed to be inside a system synthesized transaction.

In the case of WebSphere – a few million lines of Java code, we classify locks according to the following criteria: synchronized methods are data-centric; static synchronized methods and synchronized blocks are operation-centric, since their lock is not associated with any specific object instance. We randomly sampled and manually inspected the code to determine that this classification holds over a large number of cases. For the SUN JRE, we use the

MySQL 5.0.22	
LOC	1.5 million
# of Files	2336
# of CS	1275
Data-centric CS	84%
WebSphere	
# of Classes	11343
# of Sync Methods	2029
# of Static Sync Methods	546
# of Sync Blocks	2119
DC Sync Blocks (Sampled)	72%
Data-centric CS	75%
SUN JRE 1.50	
# of Classes	13081
# of Sync Methods	5337
# of Static Sync Methods	915
# of Sync Blocks	5337

Table 11.1: Estimation of the proportion of data-centric critical sections in MySQL, WebSphere and SUN JRE.

Lock	# of distinct crit. sections	# of distinct files
kernel_mutex	70	14
LOCK_thread_count	62	14
LOCK_global_system_variables	31	7

Table 11.2: Examples of globals locks with a large number of static critical sections in MySQL.

```
pthread_mutex_lock(&thd->mysys_var->mutex);
thd->proc_info=0;
thd->mysys_var->current_mutex= 0;
thd->mysys_var->current_cond=  0;
pthread_mutex_unlock(&thd->mysys_var->mutex);
```

(a) Data-centric

```
pthread_mutex_lock(&LOCK_error_log);
skr=time(NULL);
localtime_r(&skr, &tm_tmp);
start=&tm_tmp;
fprintf(stderr, ...);
fflush(stderr);
pthread_mutex_unlock(&LOCK_error_log);
```

(b) Operation-centric

Figure 11.1: Examples of typical critical sections in MySQL.

same criteria as WebSphere but did not inspect the source code to determine the nature of the synchronized blocks.

When inspecting the code, we classified critical sections as data-centric when their purpose was obvious to keep shared data-structures consistent. Critical sections were classified as operation-centric when they performed a collection of unrelated tasks, typically calls to apparently unrelated functions or methods. When in doubt, we classified critical-sections as operation-centric. Using this conservative classification, we find about 75% of the critical sections to be data-centric.

When inspecting synchronized blocks in WebSphere, we frequently observed cases where synchronized blocks were used to avoid having long critical sections by making the whole method synchronized. In those situations, the blocks were synchronized on *this*. Other common patterns were composite objects whose methods had synchronized blocks on the encapsulated object instances.

11.2 Experimental Setup

Since there are no programs written for Colorama, our evaluation consists of analyzing existing lock-based applications and estimating Colorama's potential and overheads. We analyze a variety of large, open-source, realistic multithreaded applications written in C or C++. Among them are the AOL web server, the Firefox web browser, the MySQL database server, and others. Table 11.3 lists the applications along with their number of dynamic instructions, critical sections (static and dynamic) and peak memory footprint, as they run natively on a Xeon-based multiprocessor with 8 hardware contexts.

Name	Description	# Inst (10^9)	# Critical Sec		Peak Footp (MB)
			Sta	Dyn (10^3)	
aolserver	Web server (v4.0.10)	19.5	116	1169.4	11.2
barnes	SPLASH-2 application	11.8	22	69.1	34.8
firefox	Browser (v1.5.0.1)	7.1	485	832.8	172.2
gaim	Instant msg (v2.0.0b2)	3.2	6	9.9	138.5
gftp	FTP client (v2.0.18)	1.4	173	882.0	52.9
mysql	MySQL DB (v5.0.18)	32.7	147	3302.7	545.5
tuxracer	Game (v0.5a)	10.5	74	15.7	91.7
Avg	—	12.3	146.1	897.4	149.5

Table 11.3: Multithreaded applications evaluated.

We developed a Pin-based [48] tool that profiles our applications running natively with multiple threads. The tool tracks synchronization operations and collects information such as lock acquire and release sites, lock addresses, and critical section executions and sizes. It also collects other events such as instruction counts and memory allocations and deallocations. The tool is also connected to a simulator that models a Multilevel Permissions Table for MMP [77] with Palette extensions (Figure 9.4(a)).

Synchronization operations are typically calls to multithreading libraries such as Pthreads. Many times, however, applications synchronize with indirections to pthread functions or with actual application code. An example is Tcl_MutexLock and Tcl_MutexUnlock, part of the TCL library used by *aolserver*. Our profiler can handle such cases as well.

11.3 Evaluation

We evaluate the suitability and impact of our chosen Colorama exit policy, and then examine Colorama's structure sizes and overheads.

11.3.1 Suitability of Colorama's Exit Policy

This section presents experimental evidence showing that the exit policy that we choose for Colorama in this thesis is already an informal convention largely followed by programmers of CCS code. Consequently, requiring its compliance for correct DCS code would likely be a light burden. For this experiment, we determine, for each critical section executed by the applications, whether the lock acquire and release are in the same subroutine. If they are, the section is *matched*; otherwise, it is *unmatched*.

Figure 11.2 shows the percentage of dynamic (D) and static (S) critical sections that are matched or unmatched. Recall from Table 11.3 that individual applications have 10K-3303K dynamic critical sections and 6-485 static ones. From the figure, we see that matched critical sections account for practically all the dynamic sections, and for 95% of the static ones. This supports our choice of exit policy. It shows that programmers already tend to initiate and conclude a critical section in the same subroutine.

The few unmatched cases are either special cases or are in code that is very fine-tuned for concurrency, especially in libraries. For example, in *firefox*, *gaim*, and *gftp*, all unmatched critical sections are inside the fine-tuned GTK library.

Figure 11.3 shows a representative unmatched critical section from GTK. In the figure, subroutine *g_main_dispatch* assumes that it holds lock *context*. Inside the subroutine, before the invocation of callback function *dispatch*, the code releases the lock; after the invocation, the code acquires the lock back. This structure would not be compatible with our exit policy. In this particular case, however, Colorama can handle this code without any changes because it is library code (Section 10.2).

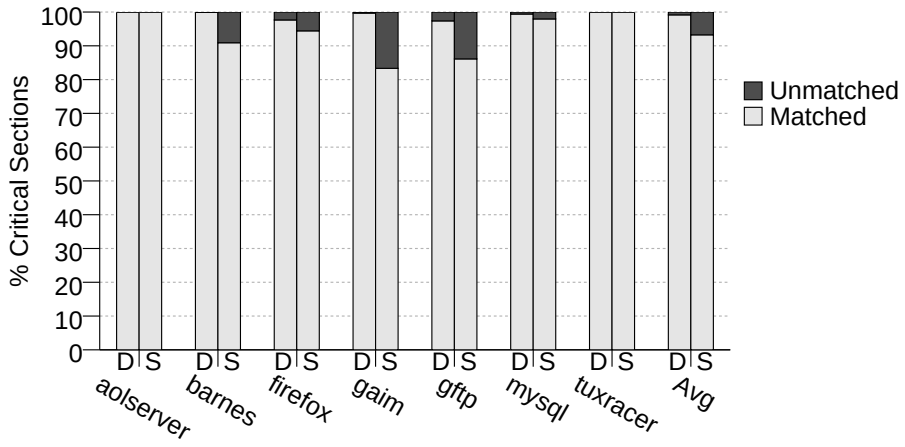


Figure 11.2: Percentage of dynamic (D) and static (S) critical sections that are matched or unmatched.

```
/* thread holds "context" lock */
g_main_dispatch (GMainContext *context)
{
  ...
  UNLOCK (context);
  ...
  need_destroy = ! dispatch (source, callback, user_data);
  ...
  LOCK (context);
  ...
}
```

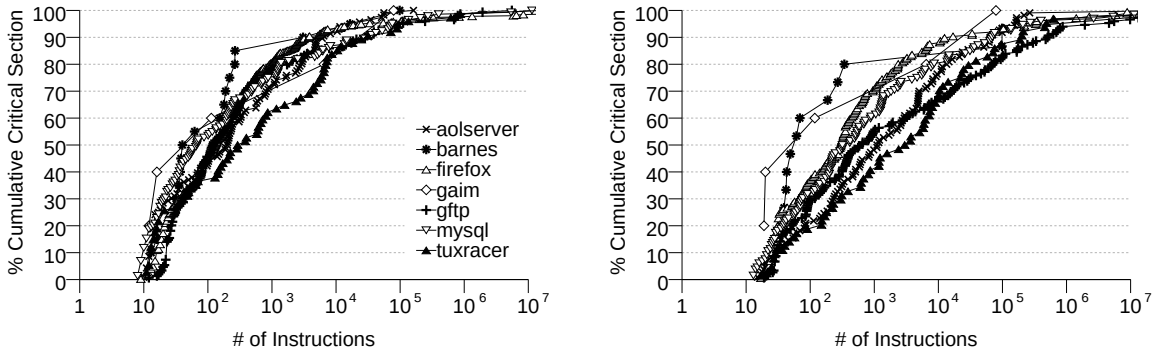
Figure 11.3: Example of code from the GTK library with an unmatched critical section.

11.3.2 Impact of Colorama's Exit Policy

The exit policy that we have chosen has two potential implications: the critical section size increases and independent critical sections may get combined in a nest. These issues typically have little or no impact in our proposed transaction-based implementation of Colorama. However, in a lock-based implementation, the first issue could increase lock contention and the second one could, under certain conditions, cause deadlock (Section 9.6.3).

To assess the first issue, we measure the average *dynamic* size of each critical section

in its lock version (from acquire to release) and in what would be its Colorama version (from acquire to subroutine return). The resulting cumulative distribution is shown in Figures 11.4(a) and (b), respectively.



(a) Acquire to release

(b) Acquire to return

Figure 11.4: Cumulative distribution of dynamic critical section size from acquire to release (a) and from acquire to subroutine return (b).

While the dynamic sizes of critical sections do increase, the average increase is not excessive. In some applications, there are a few critical sections that increase in size substantially. For example, this occurs for the sound thread in *tuxracer*. The thread acquires and releases a lock at the beginning of the game, and then runs for the duration of the game without returning from the subroutine. However, we believe that, since the Colorama programmer is required to know the system's exit policy, he will write the code to avoid lengthy critical sections.

To assess the case of independent critical sections being combined into a nest of critical sections, we measure how often multiple, independent critical sections have their entry points inside the same subroutine. These are the ones that would be combined into a nest. Figure 11.5 shows the percentage of dynamic (D) and static (S) critical sections that, because of Colorama's exit policy, would end up combining with an independent second critical section, by nesting it inside. Such instances are called *Combined*.

From the figure, we see that on average only about 1% of the dynamic critical sections

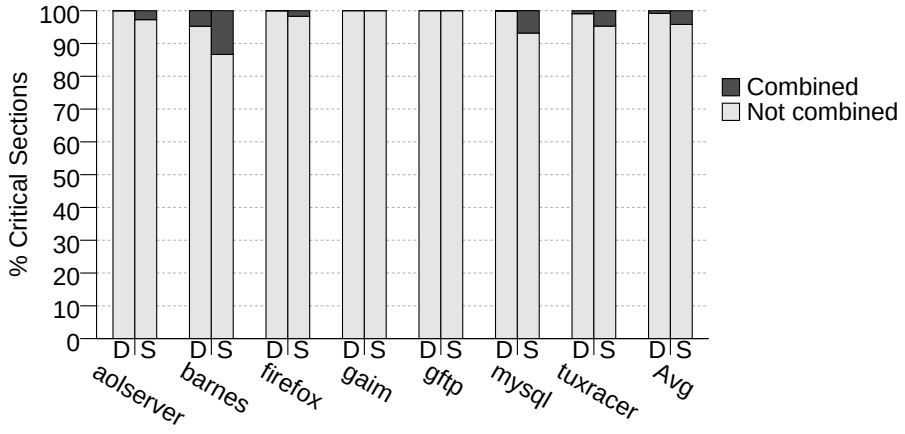


Figure 11.5: Percentage of dynamic (D) or static (S) critical sections that end up nesting a second critical section inside them.

and 4% of the static ones end up nesting a second critical section in. A detailed analysis of these (few) cases shows that the resulting order of any pair of nested locks is always the same — which eliminates the possibility of getting a deadlock. Consequently, we conjecture that the possibility of deadlock will be rare.

11.3.3 Colorama Structure Sizes

To estimate the sizes of the Colorama structures in Figure 9.1, we perform several measurements on the applications. We conservatively assume that every time an application allocates or deallocates memory, it adds or deletes, respectively, a colored region. Consequently, the number of "live" allocated regions plus the number of static data objects in the binary gives the total number of colored regions at a time. This number is shown in Column 2 of Table 11.4, and corresponds to the number of rows in the Palette. Such number ranges from 100 to nearly 1M.

We also measure the number of distinct lock addresses in each program. Such number estimates the number of different colors needed. The number is shown in Column 3. We see that programs need 100-4000 colors. From this number, we compute the number of bits in

App	Colorama Structure Sizes				Colorama Overheads	
	# of Palette Rows (10^3)	# of Colors	# of ColorID Bits	# of OCA Entries	# of Subr Calls (% Inst)	# of Inst per Col Syscall (10^3)
aolserver	0.6	141	8	39	1.9	28346.8
barnes	0.1	155	8	15	0.7	287775.4
firefox	960.1	3992	12	11	1.2	3.6
gaim	743.0	1151	11	4	1.9	1.9
gftp	15.2	874	10	6	2.5	1.9
mysql	40.7	1936	11	10	2.7	129.5
tuxracer	10.3	73	7	6	0.3	160.6
Choice	—	4096	12	64	—	—

Table 11.4: Characterization of Colorama.

ColorID. As shown in Column 4, we need 7-12 bits in the ColorID field.

Finally, to determine the number of entries in the Owned Colors Array (OCA) in Figure 9.1 (or the number of bits in the CAB and CRB registers), we need to measure the maximum number of locks held by a thread at a time. To be conservative, we measure the maximum number of locks held at a time by *all* threads combined. Such number is shown in Column 5, and ranges from 4 to 39.

The last row of Table 11.4 shows the parameters we choose for Colorama: 4K colors, 12-bit ColorIDs, and a maximum of 64 owned colors per thread. Moreover, following [77], we set the PLB to 128 entries, where each entry maps 16 words.

11.3.4 Colorama Overheads

Finally, we measure the two main Colorama overheads, namely additional instructions and additional memory space. One more overhead is the extra references to memory due to PLB misses, but these are largely the same as in the base MMP design (without Colorama) — around 8%, as quantified in [77].

Additional Instructions. For every subroutine invoked, the compiler inserts about six instructions to perform the operations shown in Figure 9.2(d). In addition, for each pointer that the subroutine takes as argument, the compiler adds one colorcheck instruction. Overall, we could assume that, on average, Colorama adds about seven instructions per subroutine

invocation. As a reference, Column 6 of Table 11.4 shows that, on average, about 1.6% of the dynamic instructions are subroutine calls.

In reality, the resulting overhead is likely to be very small. First, the added instructions are mostly register moves and loads/stores that hit in the cache — since they access the stack; they can easily fill the many unused execution slots in superscalars. Moreover, the compiler does not need to add these additional instructions for the subroutines that it can prove do not access colored data. Finally, applications often execute library code, which is not subject to this overhead.

A second source of overhead is the execution of the user-level handlers to enter and exit critical sections. However, the contribution of these instructions is very small, given the low frequency of critical section entry and exit. Such frequency is given by two times the numbers in Column 5 of Table 11.3 over the numbers in Column 3 of the same table.

Finally, Colorama also executes coloring system calls. We conservatively assume that every time the application allocates or deallocates memory, it issues one such call to add or delete a colored region, respectively. Column 7 of Table 11.4 shows the frequency of these system calls. For four applications, they are issued on average only once every 129K-288M instructions. In this case, the overhead is negligible. In three other applications, they are issued once every 2K-4K instructions. In these applications, the frequent memory allocation/deallocation is already very costly in itself. We can eliminate most of the additional cost of coloring by having the memory allocator keep pools of colored memory. As a result, there is no need to issue a system call at each of these operations.

Additional Memory Space. The large majority of Colorama's memory overhead is due to the Palette. To compute the Palette's overhead, we model in detail the MMP's Multilevel Permissions Table of Figure 9.4(a) in our simulator. We use the Mini-SST format of the entries, as suggested in [77]. We measure two memory space overheads: the one for the base MMP with permissions information (white part of the Permissions Table in Figure 9.4(a)), and the one for the Palette state only (shaded part in Figure 9.4(a)). Figure 11.6 shows

these two memory space overheads as a fraction of the application footprint. For a given bar, both these two overheads and the application footprint are the peak values for the whole application execution. For additional information, the figure models ColorID fields that range from 8 to 32 bits.

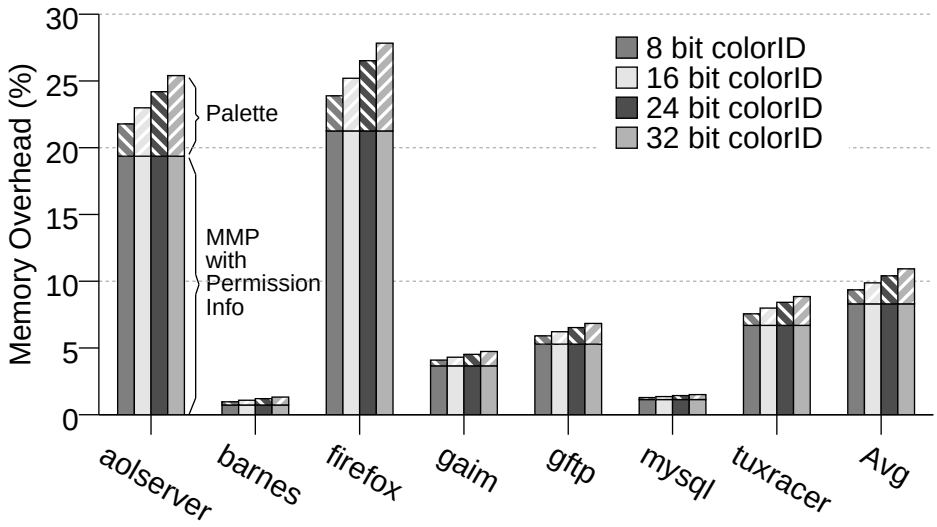


Figure 11.6: Space overhead of the base MMP and of the Palette for different ColorID sizes.

The figure shows that the Palette adds only a very modest overhead over that of the base MMP. On average, for the range of ColorIDs used, the Palette only adds 1-2.5% more space to the footprint of the application.

Chapter 12

Previous Work Related to Data-Centric Synchronization

Section 8.1 described the work that we strictly consider S-DCS, and how it differs from Colorama. To that discussion, we add that Atomic Sets [73] are what we call colors, and that Vaziri *et al.* also allow the programmer to explicitly associate external methods to an Atomic Set, which arguably breaks the pure data-centric approach.

Other systems that support a less flexible form of DCS are languages [6, 7, 35] with concurrency control based on Monitors [39]. In such languages, it is possible to specify a shared data structure and the set of procedures that are allowed to access it. The compiler will then add the necessary synchronization operations to make these procedures execute in a mutually exclusive way. The key difference is that, in Colorama, the programmer does not have to specify the procedures that touch the shared data structure. Synchronization is inferred dynamically by the hardware — an approach that is efficient, flexible, and often the only alternative when the code is hard to analyze statically or simply not available to the compiler.

Several works have associated data objects to synchronization information for a variety of purposes. For example, in Entry Consistency (EC) [8], the association is done to enforce memory consistency in a distributed shared-memory system. The programmer explicitly associates shared locations with locks. When a processor enters a critical section by acquiring a lock, the associated shared locations are made consistent. An important difference with Colorama is that in EC, the programmer explicitly marks the critical sections in the code. This makes EC code-centric, with some data-centric annotations.

Having to explicitly list the shared data associated with a critical section is a burden

to the programmer. As a result, Scope Consistency [41] improves on EC by having the software system automatically infer the shared data accessed in the scope of each critical section. Still, the programmer has to mark the critical sections.

Like Colorama, Xu *et al.* [78] try to infer critical sections, although the approach and environment is very different. They examine a post-mortem trace of memory references after a bug has been detected, and propose heuristics to infer the code that should be in critical sections. They use this information to estimate if a synchronization was missing. The Colorama hardware cannot directly use their heuristics to decide when to enter/exit a critical section because their scheme requires access to future references and to references from other processors. Moreover, their heuristics can have false positives and false negatives. However, their scheme could be usable in other DCS designs.

McKenney [52] discusses several synchronization patterns encountered in parallel programs. Particularly related to this thesis are the Code Locking, Data Locking and Data Ownership patterns. In our classification, both the Code Locking and Data Locking are code centric, the main difference between these patterns being the granularity at which the locks are placed in the program. In the Code Locking example the locks are inserted based on the procedure boundaries, while in the Data Locking example the locks are inserted in the code based on the fine-grain data structures. McKenney argues that Data Locking is a much more complex pattern to use, but worth the effort if bigger speedups are needed. In our approach, we get the benefits of Data Locking with much simpler annotations, and therefore reduced code complexity.

Other related works include: (i) programmer-specified association between code and data for static or dynamic validation of parallel programs (e.g., [68]); (ii) programmer-specified "transactional" variables in composable memory transactions [36] that provide stronger atomicity guarantees; and (iii) the lock bits associated with memory regions in the IBM 801 [18], used to support transactions on memory-mapped I/O.

Chapter 13

Conclusions

This thesis presented two main contributions aimed at improving the programmability of shared memory multiprocessors. Section 13.1 summarizes and concludes bulk operations for multiprocessors; Section 13.2 concludes the work on concurrency control with data coloring.

13.1 Bulk Operation for Multiprocessors

The contribution in this part of the thesis has been 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, TLS and high-performance SC. 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, committing by clearing a signature and decoupling consistency enforcement from the core micro-architecture.

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.

This thesis also presented Bulk enforcement of SC or BulkSC, a novel implementation of SC that is simple to implement and delivers performance comparable to RC. The idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. Then, the hardware uses signatures to enforce SC at the coarse grain of chunks rather than at the fine grain of individual memory accesses. Enforcing SC at chunk granularity can be realized with simple signature hardware and delivers high performance. Moreover, to the program, it appears as providing SC at the memory access level.

BulkSC uses mechanisms from Bulk and checkpointed processors to largely decouple memory consistency enforcement from processor structures. There is no need to perform associative lookups or to interact in a tightly-coupled manner with key processor structures. Cache tag and data arrays remain unmodified, are oblivious of what data is speculative, and do not need to watch for displacements to enforce consistency. Overall, we believe that this enables designers to conceive processors with much less concern for memory consistency issues. Finally, BulkSC delivers high performance by allowing memory access reordering and overlapping within chunks and across chunks.

We also described a system architecture that supports BulkSC with a distributed directory and a generic network. We showed that BulkSC delivers performance comparable to RC. Moreover, it only increases the network bandwidth requirements by 5-13% on average over RC, mostly due to signature transfers and chunk squashes.

Finally, the work on Bulk signatures and operations has inspired other researches to explore signatures in the context of their projects. Yen *et al.* [80] propose a log-based TM system that uses signatures to decouple transactional bookkeeping from caches. Minh *et al.* [53] propose a hybrid TM system with strong isolation guarantees by using hardware

acceleration that provides signature-based conflict detection exposed to software.

13.1.1 Other Uses of Bulk Operations

Determinism in a BulkSC System

In a *BulkSC* system, as discussed in Section 6.2.2, the arbiter imposes a total order of chunk commits, which implies in a total order of memory operations. The commit order of chunks is determined by the order that commit requests reach the arbiter and by the arbitration policies — i.e. how the arbiter state machine was designed.

The total order of memory operations is also function of what memory operations are contained in each chunk. The memory operations contained in a chunk is function of the chunking policy — i.e. where the processor chooses to place chunk boundaries.

Now lets assume a program reads input data deterministically (always reads the same input in all of its runs). When this program runs in a *BulkSC* system, the non-determinism in the total order of memory operations would come from two sources: (i) non-determinism in chunking and (ii) non-determinism is the arbiter behavior.

A *BulkSC* system could be designed to have both chunking policies and arbiter behavior to be deterministic. This would cause the total order of memory operations of a parallel program to be only function of the program's input, limiting the sources of non-determinism in the execution of a parallel program. This might be a very promising way of constructing shared-memory multiprocessors that behave deterministically.

Exposing Bulk Operations to Software

In thesis, Bulk operations have been mostly invisible to software. One possibility of further exploring signature and their operations would be to fully expose them to software.

A possible high-level architectural organization would be to have a signature register file and a collection of functional units that implement the bulk operations described in 3.2. Both

the signature register file and the functional units would be accessible via ISA extensions.

Signature registers could encode either explicit addresses directly or could automatically encode memory references performed by the processor. Other useful features would be to have external coherence requests be checked against a signature register to determine if remote memory references happen to collide with addresses encoded in signature registers. Such hardware support could be used for debugging, speculative optimizations and profiling.

Some of these uses have been explored by Tuck in his PhD thesis [71]. Tuck describes how to use bulk operations exposed to software to implement some speculative compiler optimizations such as coarse-grain redundancy elimination.

13.2 Concurrency Control with Data Coloring

This part of the thesis proposed *Colorama*, the first design of Hardware DCS (H-DCS). Colorama relies on two nimble hardware primitives to make DCS practical: one that monitors all memory accesses and one that can flexibly trigger the exit of a critical section based on a mechanism programmed in software. We have described Colorama's operation with transactions as the underlying synchronization mechanism. Moreover, we have presented Colorama's simple implementation based on MMP, its programming model and API, and its capacity to help debug conventional CCS codes. Finally, we have discussed the issues that appear in a lock-based implementation.

We justified our claim that data-centric concurrency control is the common case by inspecting the code and analyzing synchronization patterns of large software packages. We concluded that about 75% synchronization constructs are devoted to keeping shared data consistent — i.e. they are of data-centric nature. The evaluation also assessed the policy chosen in this thesis to exit a critical section at the return from the subroutine where the critical section was entered. We showed that this exit policy is already an informal convention largely followed by programmers of CCS code. Consequently, requiring its compliance

for correct DCS code will likely be a light burden at most. We also showed that the policy increases critical sections modestly on average, and rarely combines critical sections — issues largely relevant to a lock-based implementation. The evaluation also showed that, by building on top of an MMP system, Colorama requires only modest hardware resources and induces small overheads.

Overall, Colorama effectively supports general-purpose, pointer-based languages such as C/C++ and, in our opinion, can substantially simplify writing *new* parallel programs beyond transactions.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Research Report 95/7. Technical report, Western Reseach Laboratory-Compaq, 1995.

[2] S. V. Adve and M. Hill. Weak Ordering - A New Definition. In *International Symposium on Computer Architecture*, May 1990.

[3] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *International Symposium on Microarchitecture*, November 2003.

[4] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J.E.B. 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.

[5] 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.

[6] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, March 1992.

[7] J. Barnes. Introducing Ada 9X. *ACM Ada Letters*, 1993.

[8] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *IEEE International Computer Conference*, February 1993.

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

[10] C. Blundell, E. Lewis, and M. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, November 2006.

[11] H. Cain and M. Lipasti. Memory Ordering: A Value-Based Approach. In *International Symposium on Computer Architecture*, June 2004.

[12] 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*, October 2005.

[13] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural Support for Data-Centric Synchronization. In *International Symposium on High-Performance Computer Architecture*, February 2007.

[14] 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.

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

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

[17] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. 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.

[18] A. Chang and M. F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions Computer Systems*, February 1988.

[19] 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.

[20] W. Collier. Principles of Architecture for Systems of Parallel Processes. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, 1981.

[21] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *International Symposium on High-Performance Computer Architecture*, February 2004.

[22] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *International Symposium on Computer Architecture*, June 1986.

[23] C. Flanagan and S.Qadeer. A Type and Effect System for Atomicity. In *Conference on Programming Language Design and Implementation*, June 2003.

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

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

[26] 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.

[27] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *International Conference on Parallel Processing*, August 1991.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. B. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, June 1990.

[29] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[30] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *International Symposium on Computer Architecture*, May 1999.

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

[32] A. Gupta, W. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *Internaitonal Conference on Parallel Processing*, August 1990.

[33] 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.

[34] 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.

[35] P. B. Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1975.

[36] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

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

[38] M. D. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer*, August 1998.

[39] C. Hoare. Monitors - An Operating System Structuring Concept. *Communications of ACM*, 1974.

[40] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *International Symposium on Computer Architecture*, June 1996.

[41] L. Iftode, J. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Symposium on Parallel Algorithms and Architectures*, June 1996.

[42] 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.

[43] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed Early Load Retirement. In *International Symposium on High-Performance Computer Architecture*, February 2005.

[44] 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.

[45] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, July 1979.

[46] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, March 1992.

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

[48] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming Language Design and Implementation*, June 2005.

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

[50] J. F. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *International Symposium on Microarchitecture*, November 2002.

[51] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2005.

[52] P. McKenney. Selecting Locking Designs for Parallel Programs. *Pattern Languages of Program Design*, 1996.

[53] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *International Symposium on Computer Architecture*. June 2007.

[54] 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.

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

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

[57] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *International Symposium on High-Performance Computer Architecture*, February 2003.

[58] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[59] M. S. Papamarcos and J. H. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *International Symposium on Computer Architecture*, June 1984.

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

[61] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Symposium on Parallel Algorithms and Architectures*, June 1997.

[62] 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.

[63] 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.

[64] G. S. Sohi, S. E. Breach, and T. N. Vijayakumar. Multiscalar Processors. In *International Symposium on Computer Architecture*, June 1995.

[65] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[66] 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.

[67] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry. A Scalable Approach to Thread-Level Speculation. In *International Symposium on Computer Architecture*, June 2000.

[68] D. F. Sutherland, A. Greenhouse, and W. L. Scherlis. The Code of Many Colors: Relating Threads to Code and Shared State. In *Workshop on Program Analysis for Software Tools and Engineering*, November 2002.

[69] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. In *Hot Chips*, August 1999.

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

[71] J. M. Tuck. *Efficient Support for Speculative Tasking*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[72] 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 Services*, July 2005.

[73] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *Symposium on Principles of Programming Languages*, February 2006.

[74] C. von Praun, L. Ceze, and C. Cascaval. Implicit Parallelism with Ordered Transactions. In *Symposium on Principles and Practice of Parallel Programming*, March 2007.

[75] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *Symposium on Principles and Practice of Parallel Programming*, March 2006.

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

[77] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[78] M. Xu, R. Bodik, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *Conference on Programming Language Design and Implementation*, June 2005.

[79] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.

[80] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *International Symposium on High Performance Computer Architecture*, February 2007.

Author's Biography

Luis Ceze was born in São Paulo, Brazil in 1977. While growing up in São Paulo, his favorite toys were *fischertechnik* and Märklin model trains. After playing with those toys he had a natural tendency towards engineering. He got his bachelors and masters degree in Electrical Engineering from University of São Paulo in Brazil. His masters thesis was on a full-system simulation infrastructure for the IBM Blue Gene/L machine. He is currently a PhD candidate in the Department of Computer Science at the University of Illinois Urbana-Champaign (UIUC). His doctoral research has focused on computer architectures to decrease the hardware complexity and improve the programmability of multiprocessor systems.

He has co-authored over 20 papers in computer architecture, programming models, and systems. He has also participated in the Blue Gene, Cyclops, and PERCS projects at IBM. He has received awards from UIUC for research and academic accomplishments, and from IBM for contribution to the Blue Gene project. He is a recipient of the IBM PhD Fellowship.

After concluding his PhD he will join the Computer Science and Engineering Department at the University of Washington in Seattle.