BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support

W. Ahn, S. Qi, M. Nicolaides, J. Torrellas
University of Illinois at Urbana-Champaign
dahn2,sqi2,nicolai1,torrella@illinois.edu

J.-W. Lee, X. Fang, S. Midkiff
Purdue University
jaewoolee,xfang,smidkiff@purdue.edu

David Wong
Intel Corporation
david.c.wong@intel.com

ABSTRACT

A platform that supported Sequential Consistency (SC) for *all* codes — not only the well-synchronized ones — would simplify the task of programmers. Recently, several hardware architectures that support high-performance SC by committing groups of instructions at a time have been proposed. However, for a platform to support SC, it is insufficient that the hardware does; the compiler has to support SC as well.

This paper presents the hardware-compiler interface, and the main compiler ideas for *BulkCompiler*, a simple compiler layer that works with the group-committing hardware to provide a *whole-system high-performance* SC platform. We introduce ISA primitives and software algorithms for BulkCompiler to drive instruction-group formation, and to transform code to exploit the groups. Our simulation results show that BulkCompiler not only enables a whole-system SC environment, but also one that actually outperforms a conventional platform that uses the more relaxed Java Memory Model by an average of 37%. The speedups come from code optimization inside software-assembled instruction groups.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures — MIMD processors; D.3.2 [**Programming Languages**]: Language Classifications — Concurrent, distributed, and parallel languages; D.3.4 [**Programming Languages**]: Processors — Compilers, Optimization

General Terms

Algorithms, Design, Performance.

Keywords

Sequential Consistency, Atomic Region, Chunk-Based Architecture, Compiler Optimization.

1. INTRODUCTION

The arrival of multicore chips as the commodity architecture for many platforms has highlighted the need to make parallel programming easier. While this endeavor necessitates advances in all layers of the computing stack, at the hardware architecture layer it requires that multicores be designed to support programmer-friendly models of concurrency and memory consistency efficiently.

The memory consistency model specifies what values a load can return in a shared-memory multithreaded program [1]. One such model is Sequential Consistency (SC). SC mandates that the result of any execution of the program be the same as if the memory operations of all the processors were executed in some total sequential order, and those of each individual processor appear in this sequence in the order specified by its thread [15]. There is consensus that software writers prefer that the platform support SC because it offers the same simple memory interface as a multitasking uniprocessor.

For software that is well synchronized (i.e., one that does not contain data races), most systems used today support SC with high performance. This is because synchronization operations totally order those accesses from different threads that, if overlapped, could result in non-intuitive return values for loads. Unfortunately, much current software, ranging from user applications to libraries, virtual machine monitors, and OS, has data races — either by accident or by design. For these codes, SC is not provided. Moreover, as more beginner programmers attempt parallel programming on multicores in the next few years, the number of codes with data races may well increase.

1.1 Benefits of Supporting SC

Devising a platform that supports SC with high performance for *all* codes — including those with data races — would have four key benefits. The first one is that debugging concurrent programs would be easier. This is because the possible outcomes of the memory accesses involved in the bug would be easier to reason about, and the debugger could in fact *reproduce* the buggy interleaving.

A second benefit stems from the fact that existing software correctness tools almost always assume SC — for example, Microsoft's CHESS [19]. Verifying software correctness under SC is already hard, and the state space balloons if non-SC interleavings need to be inspected as well. In the next few years, software correctness verification tools are expected to play a larger role. Using them in combination with an SC machine would make them most effective.

A third benefit of SC is that it would make the memory model of safe languages such as Java easier to understand and verify. The need to provide safety guarantees and enable performance at the same time has resulted in an increasingly complex and unintuitive memory model over the years. A high-performance SC memory model would trivially ensure Java's safety properties related to memory ordering, and improve its security and usability.

Finally, some programmers want to program with data races to obtain high performance. This includes, for instance, writers of OS and virtual machine monitors. If the machine provided SC, the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'09, December 12–16, 2009, New York, NY, USA.
Copyright 2009 ACM 978-1-60558-798-1/09/12 ...$10.00.

risk of introducing bugs would be reduced and the code portability enhanced.

1.2 Goal of the Paper and Contributions

From this discussion, we argue that supporting SC is a worthy goal. Recently, there have been several proposals for hardware architectures that support high-performance SC [3, 4, 6, 11, 12, 29, 32]. Some of these architectures support SC all the time by repeatedly committing groups of instructions atomically — called chunks in BulkSC [6], transactions in TCC [12], or implicit transactions in checkpointed multiprocessors [29]. Each instruction group executes atomically and in isolation, generating a total commit order of chunks and, therefore, instructions, in the machine. Such properties guarantee SC. Moreover, thanks to operating in large instruction groups, the overheads of supporting SC are small. Conceivably, a similar environment can be attained with a primitive for atomic region execution such as that of Sun's Rock [7], if it is invoked continuously.

Unfortunately, for a platform to support SC, it is *not enough* that the hardware support SC; the software — in particular, the compiler for programs written in high-level languages — *has to support SC as well*. For this reason, there have been several research efforts on compilation for SC [13, 28, 31]. Such efforts have sought to transform the code to satisfy SC on conventional multiprocessor hardware. The results have been slowdowns — often significant — relative to the relaxed memory models of current machines.

Remarkably, with the group-commit architectures, we have an opportunity to develop a high-performance SC compiler layer. Since the hardware already supports high-performance SC, all we need is for the compiler to drive the group-formation operation, and adapt code transformations to it. With the combination of hardware and compiler, the result is a *whole-system high-performance SC platform*. Furthermore, since the hardware guarantees atomic group execution, the compiler can attempt more aggressive optimizations than in conventional, relaxed-consistent platforms. The result is even *higher performance* than current aggressive platforms.

This paper presents the hardware-compiler interface and the main ideas for a compiler layer that works in the BulkSC architecture (as a representative of the group-commit architectures) to provide whole-system high-performance SC. We call our compiler algorithm *BulkCompiler*. Our specific contributions include: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms with Java programs on a simulated BulkSC architecture.

Our results use Java applications modified with our compiler algorithms and compiled with Sun's Hotspot server compiler [22]. A whole-system SC environment with BulkCompiler and simulated BulkSC architecture outperforms a simulated conventional hardware platform that uses the more relaxed Java Memory Model by an average of 37%. The speedups come from code optimization inside software-assembled instruction chunks.

This paper is organized as follows: Section 2 gives a background; Sections 3 and 4 describe BulkCompiler and how it manages the chunks; Sections 5 and 6 evaluate the system; Section 7 assesses the results, and Section 8 discusses related work.

2. BACKGROUND

We describe the BulkSC architecture and the current approaches for compiler-driven enforcement of SC.

2.1 BulkSC: High-Performance SC Hardware

In the BulkSC multiprocessor [6], as a processor executes a thread, it automatically breaks the instruction stream into chunks and commits each chunk atomically. A *Chunk* is a group of *dynamically* contiguous instructions — 2,000 in the current implementation. This "chunked" mode of execution and commit is a hardware-only mechanism, which is invisible to the software running on the processor.

Each chunk executes on the processor *atomically* and *in isolation*. This means that none of the actions of the chunk are made visible to the rest of the system (other processors and main memory) until when the chunk commits. Moreover, if the chunk reads a location and, before it commits, a second chunk in another processor that has written to the same location commits, then the local chunk gets squashed and has to re-execute. Atomic chunk execution is supported by buffering in the L1 cache the state that the chunk is generating. Moreover, as the chunk executes, a Bloom filter automatically encodes in a R and W signature, the memory addresses read and written, respectively. After the chunk completes, the hardware sends W to an arbiter, which forwards it to other processors. In the other processors, W is intersected with the local signatures. A non-null result indicates an overlap of addresses, which causes the chunk in that processor to get squashed and restarted.

Since chunks execute atomically and in isolation, commit in program order in each processor, and the arbiter globally orders their commit, BulkSC supports SC at the chunk level — and, as a consequence, SC at the instruction level.

This is a high-performance SC implementation because the hardware can reorder and overlap all memory accesses within a chunk — except, of course, those that participate in single-thread dependences. In particular, synchronization instructions induce no reordering constraint. Indeed, *fences* inside a chunk are *transformed into no-ops* by the hardware. Their functionality — to delay execution until certain references are performed — is useless since, by construction, no other processor will observe the actual order of instruction execution within a chunk. Moreover, a processor can also overlap the execution of consecutive chunks [6].

2.2 Algorithm for Generating Chunks

In BulkSC, the hardware finishes the current chunk and starts a new one when the number of dynamic instructions executed exceeds a certain threshold that we call *maxChunkSize* (e.g., 2,000 instructions). There are, however, some events that affect the regular generation of chunks. Table 1 lists these events and, under *Actions in BulkSC*, the actions taken [6]. For example, when the write set of the chunk is about to overflow the cache, the hardware commits the current chunk at this point and starts a new chunk. The last column of the table will be discussed later.

2.3 Compiler-Driven Enforcement of SC

A compiler can take programs with potential data races and transform them to enforce SC even on a machine that implements a relaxed memory consistency model [13, 28, 31]. The general idea is to identify the minimal set of ordered pairs of memory accesses that should not be re-ordered, and then (1) insert a fence along every path between the first and second access in each pair, and (2) prohibit the compiler from performing any transformation that reorders any such pair.

The compiler analysis needed involves first performing Escape analysis [28], which determines which loads and stores may refer to memory locations accessed by multiple threads. Then, May-happen-parallel (or Thread-structure) analysis [20, 28] determines which memory accesses can happen in parallel. Based on these, Delay Set analysis [26] determines which of the shared accesses should not be reordered within a thread.

Event	Actions in BulkSC	Actions with BulkCompiler Inside Atomic Region
maxChunkSize instructions executed	The hardware commits the current chunk and starts a new chunk	No action
Cache overflow	The hardware commits the current chunk at this point, and starts a new chunk	The hardware squashes the current chunk and restarts it at the Safe Version point
Data collision with remote chunk	The hardware squashes the chunk and re-executes it. If the chunk is squashed *M* times, then the chunk also reduces its size to minimize collisions	Same as in under BulkSC. However, if the chunk size has to be reduced, restart the chunk at the Safe Version point
Exceptions (including system calls)	When the code wants to perform an uncacheable access, the hardware commits the current chunk at this point, performs the uncached operation, and starts a new chunk	When the code wants to perform an uncacheable access, squash the chunk and restart it at the Safe Version point. Do not set up an atomic region to include uncacheable accesses
Interrupts	The hardware completes the current chunk and then processes the interrupt in a new chunk(s)	The hardware squashes the current chunk, processes the interrupt, and then restarts the initial chunk under an atomic region again

Table 1: Events that affect chunk generation.

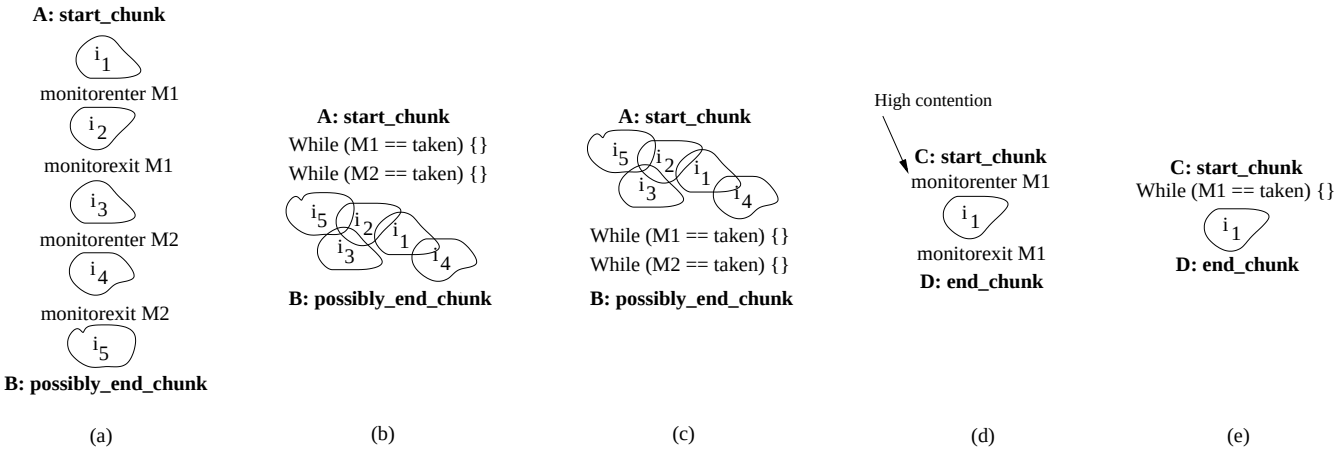


Figure 1: Compiler-driven chunking for high performance SC. In the figure, each i_j represents a set of instructions.

Unfortunately, the compiler analysis required is very costly both in runtime and in implementation effort — in part because every step needs interprocedural analysis. Moreover, all three existing implementations [13, 28, 31] report noticeable slowdowns relative to execution of the application under the relaxed model — in some cases, applications become several times slower. Our paper's goal is to deliver SC with even higher performance than current relaxed models.

3. BULKCOMPILER FOR SC

We submit that an architecture with continuous group commit such as BulkSC [6], TCC [12], or Checkpointed Multiprocessors [29] can potentially deliver whole-system (hardware plus software) SC at a higher performance than conventional machines deliver a relaxed memory consistency model. This is because, if the compiler drives chunk formation appropriately, the atomicity guarantee of chunks can enable many compiler optimizations inside the chunk.

In particular, we focus on multiprocessor-related issues. We observe that synchronization and fences can substantially hurt the performance of conventional relaxed-consistency machines. At the same time, synchronization-aware chunk formation can eliminate some of these problems, and further enable conventional compiler optimizations that improve performance.

In this section, we discuss the main ideas, the new instructions added, and the basics of the algorithms in *BulkCompiler* — our compilation layer for group-commit architectures. In a later section (Section 7), we briefly discuss how we can also improve the performance of relaxed memory consistency models in these architectures and enable new compiler optimizations.

3.1 Main Ideas

A compiler for a group-commit architecture should select the chunk boundaries so that they (1) maximize the potential for compiler optimization and (2) minimize the chance of chunk squash. Since the design space is large, this paper focuses on the multiprocessor related issues of synchronization and fences. In this area, BulkCompiler relies on one idea to maximize compiler optimization and one to minimize squashes.

3.1.1 Maximizing Compiler Optimization

To maximize compiler optimization, BulkCompiler identifies *low contention* critical sections (which are mostly in the form of synchronized blocks in Java). Then, it includes one or several of them and their surrounding code in the same chunk (Figure 1(a)). After this, each acquire operation (*monitorenter* instruction in Java bytecode) is replaced with a spinning loop, which checks if the synchronization variable is taken using *plain loads*. Moreover, all the release operations (*monitorexit* in Java bytecode) are removed. Next, we move the spinning on the locks with plain loads to the top of the chunk — subject to data and control dependences — to prepare the code for compiler optimization better. Finally, with the synchronizations removed, we let the compiler aggressively reorder and optimize the code inside the chunk. The resulting code is shown in Figure 1(b), where the overlapping sets of instruction denote the

Instruction	Functionality
beginAtomic PC	Finishes the current chunk, triggers a register checkpoint in hardware, and starts a new chunk. It takes as argument the program counter (PC) of the entry point to the *Safe Version* of the code, which will be executed if the chunk needs to be chopped into smaller chunks.
endAtomic&Cut	Finishes the current chunk and changes the mode of chunking from software-driven to hardware-driven. The hardware will start a new chunk next.
endAtomic	Changes the mode of chunking from software-driven to hardware-driven, enabling the hardware to finish the current chunk when it wants to (e.g., when the chunk size reaches *maxChunkSize*).
squashChunk	Squashes the current chunk and restarts it at the Safe Version. It involves clearing the BulkSC signatures, invalidating the cache lines written by the chunk, and restoring the checkpointed register file.
cutChunk	Finishes the current hardware-driven chunk, inducing the hardware to start a new one. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region.

Table 2: Instructions added so that the compiler manages the chunking.

effect of compiler optimization. Note that checking all the locks at the beginning of the chunk may slightly reduce concurrency. However, since we apply this transformation to low-contention critical sections, such effect is insignificant.

Since the chunk will be executed atomically, there is no need to acquire and release a lock. However, the chunk still needs to read the locks with plain loads, to check if any lock is taken. A lock can be taken if another thread, after failed attempt(s) to execute its own chunk atomically, reverted to a (non-speculative) *Safe Version* of the code, where it grabbed the lock. We will see in Section 3.4 that every atomic region has a corresponding Safe Version, where any locks are acquired and released explicitly. This is the same approach followed by the Speculative Lock Elision (SLE) algorithm [23] and its implementation in the Sun Rock [9].

If any of the locks is taken, the code spins. When the owner of the lock commits the lock release, the spinning chunk will observe a data collision on the spinning variable. At that point, it will be squashed and re-started.

By eliminating the synchronization operations, this transformation improves performance in two ways. First, the processor avoids performing the costly synchronization operations, replacing acquires with the much cheaper loads. More importantly, however, is that this transformation eliminates the constraints on instruction reordering imposed by synchronization instructions. Indeed, even under current relaxed memory models, compilers neither move instructions across synchronization operations nor allocate shared data in registers across them. This disables many instances of conventional optimizations such as register allocation, common subexpression elimination, loop invariant code motion, or redundant code motion, to name a few. After we remove the synchronization operations, a conventional compiler can reorder instructions and perform all of these optimizations.

We can place the spinning on the locks with plain loads at the end of the chunk, after all the work is done (Figure 1(c)). This approach makes a difference when one or more locks are taken by other processors and, therefore, the chunk will eventually be squashed. In this case, having the spinning at the end of the chunk can enable prefetching of read-only data for the chunk re-execution. However, it may also cause exceptions resulting from accessing data of a critical section while another processor is also accessing it. Overall, since we apply this transformation to low-contention critical sections, these effects are not very significant.

Finally, this transformation is especially attractive in Java programs, which is the environment examined in this paper. This is because Java programs have many low-contention critical sections in the form of synchronized methods — often in thread-safe Java libraries. The synchronized blocks in these methods are compiled into Java bytecode using the *monitorenter* and *monitorexit* bytecode instructions surrounding the code in the block.

3.1.2 Minimizing Squashes

The second idea in BulkCompiler is to minimize squashes by identifying *high-contention* critical sections and tight-fitting a chunk around it (Figure 1(d)). As in the previous transformation, *monitorenter* is replaced with a loop that checks if the lock is taken using plain loads. *Monitorexit* is removed (Figure 1(e)). Tight-fitting the chunk reduces the chances that different processors collide on this critical section, and also reduces the number of wasted instructions per squash. It also enables processors to hand over access to popular critical sections to other processors sooner, since chunks commit sooner.

Even after all these transformations, chunks created by the compiler can collide at runtime — either on the synchronization variable or on another variable. In this case, they retry as per the default BulkSC execution. However, there are events that require reducing the size of the chunk, such as a cache overflow or performing an uncached memory access. Reducing the chunk size could lead to non-SC executions if the broken chunk exposes reordered references to shared data. To prevent this, BulkCompiler also creates the Safe Version of the code mentioned before. The Safe Version does not reorder references to shared variables and includes the *monitorenter* and *monitorexit* instructions.

Overall, with these changes on top of high-performance SC hardware, we target a performance higher than that attained with the relaxed Java Memory Model on conventional hardware, while providing whole-system SC.

3.2 New Instructions Added

Table 2 shows the instructions added to enable the compiler to manage the chunking. The principal ones are *beginAtomic*, which marks the beginning of an atomic region, and *endAtomic&Cut* or *endAtomic*, which mark the end. *BeginAtomic* causes the BulkSC hardware to finish the current chunk and start a new one. It also creates a register checkpoint to revert to if the chunk is squashed. The instruction takes the program counter (PC) of the entry point to the Safe Version of the code for the chunk. When the atomic region is squashed, depending on the reason for the squash, the hardware returns execution to either the *beginAtomic* instruction or the entry point to the Safe Version.

EndAtomic&Cut terminates the current chunk and then lets the BulkSC hardware take over the chunking — the hardware will start a new chunk next. *EndAtomic* simply lets the BulkSC hardware take over the chunking. This means that the current chunk may continue executing until a total of *maxChunkSize* instructions since *beginAtomic* have been executed. When a chunk is executing within the *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) instruction pairs, reaching the *maxChunkSize* instruction count does not cause chunk termination. Overall, with these primitives, we surround the groups of low-contention synchronized blocks as in Fig-

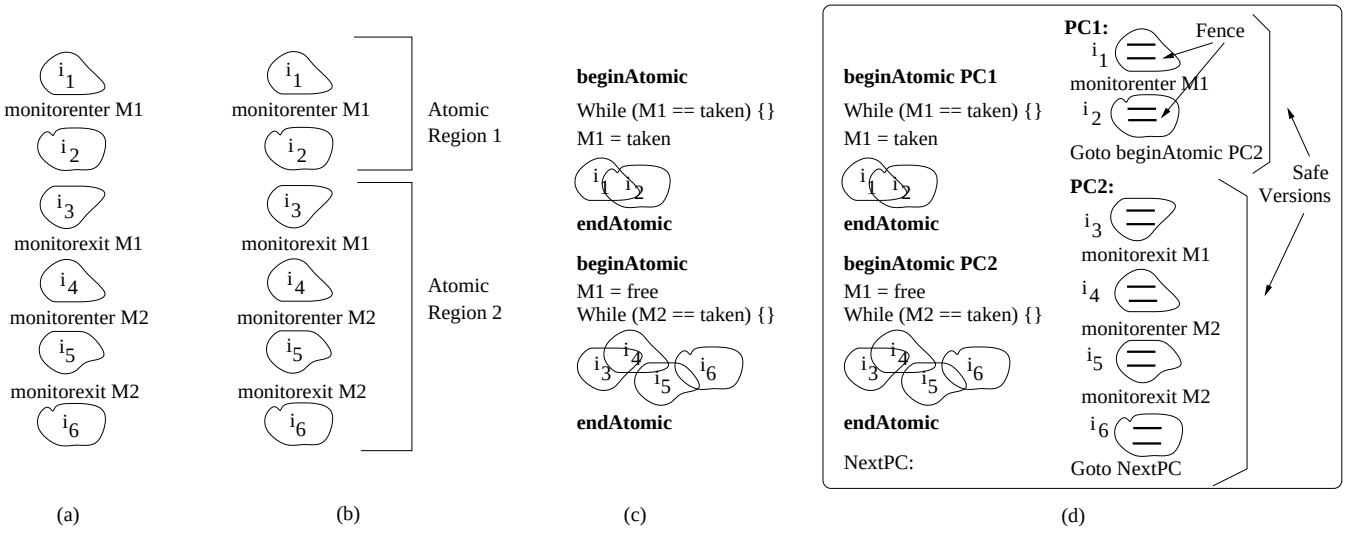


Figure 2: Transforming a large code section. In the figure, each i_j represents a set of instructions.

ure 1(b) with *beginAtomic* at point *A* and *endAtomic* at point *B*; we surround the high-contention synchronized blocks as in Figure 1(e) with *beginAtomic* at point *C* and *endAtomic&Cut* at point *D* in the figure.

To the compiler, *beginAtomic* has acquire semantics, which means that it cannot move any escaping reference (i.e., reference to shared data) that follows *beginAtomic* to before it. *EndAtomic&Cut* and *endAtomic* have release semantics, and the compiler cannot move any escaping reference that precedes them to after them.

The table shows two more instructions, called *squashChunk* and *cutChunk*. The former squashes the current chunk and restarts at the Safe Version. It can be used for speculative compiler optimizations, which sometimes require a rollback after discovering that they have performed an illegal transformation (e.g., [21]). The *cutChunk* instruction simply finishes the current hardware-driven chunk, inducing the hardware to start a new chunk. It has no effect if found inside a *beginAtomic* to *endAtomic&Cut* (or *endAtomic*) region. Note that if, dynamically, *endAtomic&Cut*, *cutChunk*, or potentially *endAtomic* are immediately followed by *beginAtomic*, the latter does not start a second chunk beyond the one that the hardware is starting.

3.3 Difference to Transactional Memory

To understand our transformations, it is useful to compare them to Transactional Memory (TM). The main goal of TM is enhancing concurrency; the main goal of our transformations is enhancing the performance of each thread through compiler optimization while preserving SC. However, since we focus on optimization opportunities afforded by synchronizations, our use of an SLE-like algorithm also enhances concurrency, especially in high-contention critical sections.

To see the difference between the two goals, consider a synchronized block that is too large for the hardware to provide atomicity. Unlike TM, BulkCompiler still benefits from splitting the code into two atomic regions. This is seen in Figure 2(a), which shows code with two synchronized blocks protected by locks M1 and M2. Assume that BulkCompiler estimates that the code in the M1 block has a footprint that amply overflows the cache. Further, assume that it estimates that the code before the M1 block (i_1) could be optimized together with the code inside the block. In this case, it partitions the code into two atomic regions that it estimates fit in the cache (Figure 2(b)): one that executes i_1 and the beginning of the first block, and another that executes the rest of the code.

BulkCompiler relies on the hardware guarantee that each region executes atomically. It transforms the code as shown in Figure 2(c): synchronization operations become plain accesses and the code is aggressively reordered and optimized. In particular, in the first atomic region, *monitorenter* is replaced with a spinning loop, which checks if the lock is taken using plain loads. If the lock is free, the code sets it to taken. If the chunk eventually finishes and commits, this lock update will be made visible; however, the chunk may be squashed before committing by the commit of another chunk that also set the lock. On the other hand, if the lock was not free, the code spins and will not commit. The chunk will eventually get squashed, either when the thread is pre-empted from the processor or when the chunk that releases the lock commits.

In the second atomic region, *monitorexit M1* simply becomes a plain write to the lock variable to release it. If the chunk commits, the write will be visible to the rest of the processors. Note that lock variable M1 has to be explicitly written as taken or freed, although the writes can be plain stores. This is because, since the synchronized block is now split into two regions, atomicity is no longer guaranteed and we have to rely on the value of the variable to prevent illegal interleavings. Finally, the accesses to *M2* are replaced with a spinning loop on *M2* with plain loads as described before. Overall, in all cases, the rest of the code is heavily optimized and the system satisfies SC.

3.4 Safe Version of the Atomic Region Code

It is possible that an atomic region gets squashed. Recall that Column 2 of Table 1 showed the events that affect chunks in the original BulkSC architecture. The last column of the table shows how we slightly change the BulkSC hardware so that it guarantees the atomicity of atomic regions.

First, inside an atomic region, the chunk is prevented from finishing when the number of instructions reaches past *maxChunkSize*, to guarantee that the entire atomic region does in fact commit atomically. Second, since this requirement can result in long atomic regions, we want to process interrupts as soon as they are received — rather than waiting until the current chunk completes. Conse-

quently, on reception of an interrupt, the current chunk is squashed, the interrupt is processed, and then the initial chunk is restarted from the beginning — using the checkpoint from *beginAtomic*.

Finally, to guarantee the atomicity of atomic regions, events that previously triggered a chunk squash may need to be handled differently. These events include (i) cache overflows, (ii) uncacheable accesses in exceptions (which include system calls), and (iii) data collisions with a remote chunk. How we handle these events largely depends on whether the event will (likely) repeat after the chunk is squashed and restarted.

The events that are unlikely to repeat are most data collisions. In this case, the atomic region is squashed and then re-executed from the beginning. The events that repeat are cache overflow, uncacheable accesses in exceptions, and repeated data collisions on the same chunk in pathological cases. Some cases of uncacheable accesses can be avoided by not including problematic system calls inside atomic regions. However, the rest of the events are largely unpredictable and hard to avoid. The atomic region cannot be simply squashed and re-executed since it will be squashed again.

To make progress in these cases, we would have to commit a downsized chunk —- i.e., the code up until we cause the cache overflow, or reach the uncacheable access or the access that causes the collision. However, this would break the atomicity of the chunk and, potentially, expose inconsistent or non-SC state. Consequently, to address these cases, a Safe Version of the code is generated for each atomic region. This safe code does not rely on atomic execution to preserve SC. If the atomic region needs to be truncated for any of the "repeatable" reasons, the chunk is squashed and execution is transferred to the PC of the Safe Version entry point — as given in the *beginAtomic* instruction.

The Safe Version of the code acquires and releases locks explicitly. Moreover, it also has to satisfy SC. Therefore, BulkCompiler conservatively identifies all the escaping references in the code using the algorithm in [16]. Then, it adds a fence at the beginning of the Safe Version code, and after every escaping reference. The fences prevent the compiler from reordering the escaping accesses — and hence ensure SC at a performance cost. The analysis of Section 2.3 could keep the overheads to a minimum. Figure 2(d) shows the final code for the example.

Fortunately, part of this performance loss is transparently recovered by the chunking hardware. Specifically, as the BulkSC hardware executes the Safe Version code with hardware-driven chunks, fences are *no-ops* (Section 2.1). The accesses that fall in the same chunk will be overlapped and reordered by the hardware, irrespective of the presence of the fences. Note also that, since Safe Versions are rarely executed, they will not hurt the instruction cache through code bloat noticeably.

4. ALGORITHM DESIGN

In this section, we describe the algorithms that we use and some of the corner cases encountered.

4.1 Inserting Atomic Regions

At the highest level, our algorithm desires to have all escaping references contained in atomic regions, and for each region to be as large as possible to expose the maximum number of optimization opportunities. Doing this naively, however, will lead to excessive squashing of atomic regions due to conflicts or cache overflow, and difficulty in generating code for the Safe Versions of the regions.

The algorithm that we use is shown in Figure 3. This algorithm is applied to each method in turn. Prior to actually selecting atomic regions, the algorithm performs aggressive inlining, escape analysis [16], and loop blocking. Inlining reduces the impact of using an intraprocedural algorithm for selecting atomic regions. Escape analysis identifies the escaping references in the method, namely the references to objects that may be accessed by two or more threads. These references should be enclosed in atomic regions. Finally, loop blocking transforms inner-most loops into a loop nest, with a constant bound on the iteration count of the innermost loop. This allows the innermost loop to be enclosed in an atomic region that fits in the cache. Loops not containing any escaping references need not be blocked.

1. Perform aggressive inlining.
2. Perform escape analysis and mark escaping references.
3. Block inner-most loops that have escaping references.
4. Traverse code while enclosing each escaping reference in an atomic region.
5. Expand each atomic region r that is immediately control dependent on statement c. We enclose adjacent statements s while all the following hold:
   a. s is control equivalent to r. If s is not control equivalent to r, then:
      i. if s is inside the c control structure, expand r to contain the code from s to P_s (the post-dominator of s). The same applies if P_s is encountered first.
      ii. if $s = c$, first expand r downwards until P_c (the post-dominator of c), and then also add c to r. The same applies if P_c is encountered first.
   b. the estimated footprint of r fits in the cache.
   c. s is not in a highly-contended synchronized block that does not contain r.
6. Generate the Safe Version for all the atomic regions.

Figure 3: Algorithm that inserts atomic regions in a method.

The algorithm then begins a traversal of the code, and each escaping reference is placed into an atomic region. After all escaping references are enclosed in atomic regions, a second pass is made to expand atomic regions and merge them where necessary. In the second pass, each atomic region is visited in turn. When an atomic region is visited, it is expanded to enclose code before and after the atomic region, with limits on this expansion as described shortly. If the expansion of an atomic region r_i encounters another atomic region r_j, r_j is merged into r_i, forming a single atomic region.

Three conditions need to hold during this expansion process. The first one is that the atomic region must begin and end at control equivalent points. Let c be the statement on which region r containing escaping reference e is immediately control dependent. This condition is easily satisfied when the statement s encountered while expanding region r is control equivalent to r. However, if it is not control equivalent, care must be taken. Specifically, (1) if s is inside the c control structure and the code from s to P_s (the post-dominator of s) is small enough so that s to P_s fits in the region, then s to P_s is added to r. The same applies if P_s is encountered instead of s. Moreover, (2) if $s = c$, then r is first expanded to cover all statements between c to P_c (the post-dominator of c) such that all statements control-dependent on c are inside r, and then c is also added to r. The same applies if P_c is encountered instead of c.

The second condition is that the estimated footprint of the atomic region fits in the cache. A model is used to estimate the contribution of each statement to the footprint. However, the available footprint is assumed exceeded if the algorithm attempts to (1) expand the atomic region into a loop other than the innermost loop around the

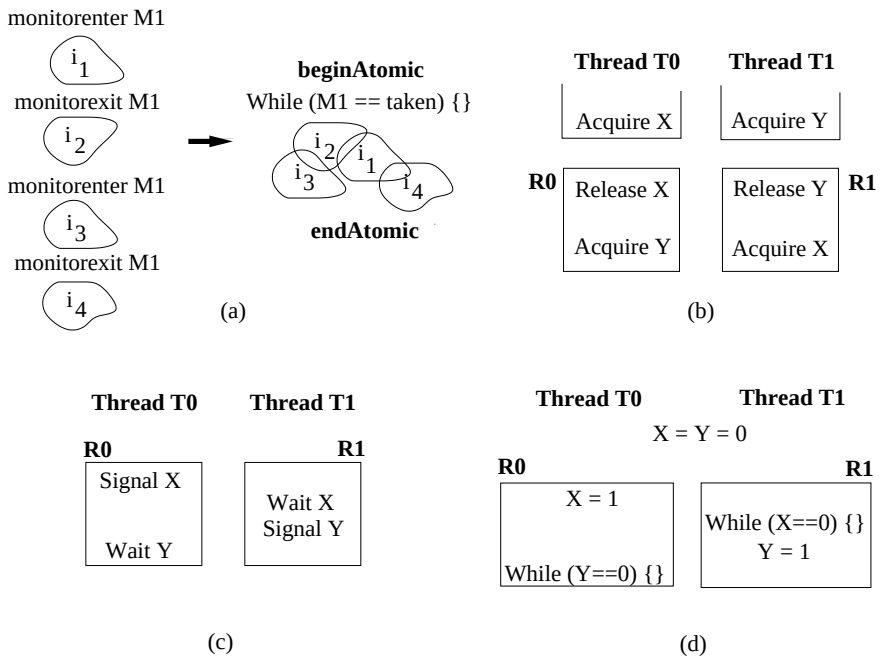


Figure 4: Examples of chunks with synchronization operations.

escaping reference e, or (2) include in the atomic region a non-inlined method call.

The third condition is that atomic regions will not expand to contain statements within a highly-contended synchronized block unless the escaping reference e is in that block. If e is in a highly-contended block, the region will at most be expanded to cover the highly-contended synchronized block.

The resulting atomic regions start and end at control equivalent parts of the program. This ensures that all atomic region starts have a corresponding atomic region end, regardless of the path taken by the program when executing. It also simplifies the generation of the code for Safe Versions.

Finally, Safe Versions of the regions are formed by duplicating the block of code in the atomic regions. A fence is placed at the beginning of the Safe Version code and after every escaping reference, to ensure that the compiler does not reorder escaping references.

4.2 Lock Compression and Region Nesting

When an atomic region contains multiple synchronized blocks protected by the same lock, the algorithm introduces a single check for the lock variable (Figure 4(a)). We call this scheme *Lock Compression*.

It is possible that the code contains nested atomic regions. At runtime, our chunking hardware flattens them out, and considers them just one large atomic region. To do this, the hardware keeps a nesting-level counter, and the chunk ends only at the outermost *endAtomic&Cut* or *endAtomic*. Moreover, when a squash is triggered, the outermost atomic region is squashed and, if appropriate, its Safe Version is invoked.

4.3 Visibility with Synchronizations

When our algorithm produces an atomic region with accesses to multiple synchronization variables, there may be interactions between threads that cause problems of *Visibility*. As an example, the problem occurs when an atomic region in Thread *T0* releases variable *X* and acquires variable *Y*, while an atomic region in Thread *T1* releases variable *Y* and acquires variable *X*. This is shown for regions *R0* and *R1* in Figure 4(b). For simplicity, the figure shows acquire and release operations — in practice, our algorithm will have replaced them with plain memory accesses to the variables. In the figure, Region *R0* cannot complete and make its release of *X* visible to *R1* because it is spinning on *Y*, which *T1* holds. *R1* is in a symmetrical situation. The result is deadlock, as both threads are spinning on the acquires.

The problem is not limited to a pattern where both threads first release a variable and then acquire a second one. It also occurs when there is a *handshake* pattern between two threads. This pattern is shown in Figure 4(c), using Signal and Wait synchronization operations. Again, we show these operations for simplicity, although our algorithm uses plain accesses. In the figure, Region *R0* signals synchronization variable *X* (effectively a release) and then waits on *Y* (effectively an acquire), while *R1* waits on *X* and then signals *Y*. Both threads end up spinning on the waits, unable to complete the regions.

These visibility problems do not occur with the hardware-driven chunks of BulkSC [6]. This is because such chunks complete as soon as *maxChunkSize* instructions are executed, rather than when a certain static instruction is reached. In both examples, the threads would spin in the acquires (or in the waits) until they reach *maxChunkSize* instructions. At that point, they would finish the chunks, making the two releases (in Figure 4(b)) or the signal to *X* (in Figure 4(c)) visible.

Similar visibility problems have been observed by proposals that integrate locks and transactions [24, 33] and by discussions of transactional memory atomicity semantics [18]. Ziarek *et al* [33] propose to solve the deadlock problem by detecting that two transactions are not completing, squashing them, and executing lock-based versions of the code. The authors state that these cases happen rarely.

BulkCompiler uses a similar approach, which is detailed in Section 4.4 and is simpler to implement. However, the problem with "unpaired" synchronization shown in Figure 4(b) cannot occur for

high-contention critical sections because BulkCompiler tight-fits the atomic region around the section. For low-contention critical sections, an unpaired synchronization may be lumped with other access(es) to synchronization variable(s) within a single atomic region. However, because of the low contention for the synchronization variables, the probability of an interleaving that causes deadlock is very low. An alternative design is to have the compiler disable the creation of such atomic regions.

4.4 Visibility with Data Races

If the code is not properly synchronized, data races may produce the deadlock-prone access patterns discussed above. For example, Figure 4(d) shows data races that create the handshake pattern. In this case, the compiler may be unable to detect the possibility of deadlock — except, perhaps, at the cost of expensive and conservative *Must-alias* analysis.

To handle this case and other deadlocks at runtime, BulkCompiler relies on detecting that two chunks are not completing, squashing them, and then triggering the execution of their Safe Versions. Note that, in our environment, detecting that chunks are not completing is easy. Rather than measuring wall-clock time, we count the number of completed instructions — which is needed by the BulkSC hardware anyway. If this number is very high, the processor is likely spinning on a tight loop. At that point, the spinning chunks are squashed and the Safe Versions executed. One option for chunks that suffer frequent timeouts is recompilation.

5. EXPERIMENTAL SETUP

5.1 Compiler and Simulator Infrastructure

Our evaluation infrastructure uses two main components: the Hotspot Java Virtual Machine (JVM) for servers [22] from Sun Microsystems and a Simics-based [30] simulator of the BulkSC architecture [6]. Hotspot is an aggressive commercial-grade compiler with extensive support for just-in-time compilation and adaptive optimization. It is included in OpenJDK7 [27]. We use Hotspot to compile both the unmodified applications for a conventional architecture, and the applications modified with the BulkCompiler algorithms for a BulkSC architecture. We report the difference in performance.

We apply the algorithm described in Section 4.1 to Java source code using a profile-driven infrastructure that currently requires substantial hand-holding. We are in the process of automating the infrastructure. Since we are instrumenting at the Java source code level, we cannot directly insert our assembly instructions of Section 3.2. Instead, we use the JNI (Java Native Interface) to wrap the instructions in Java methods — at the cost of some overhead.

The resulting modified source is compiled to bytecode, and then run on the Hotspot JVM. The Hotspot JVM executes on top of a full-system execution-driven simulator built using Simics [30]. The simulator uses the x86 ISA extended with the BulkCompiler instructions. The simulator models a BulkSC multiprocessor [6], including the chunk-based speculative execution, checkpointing, chunk squash and rollback, signature operation, and the extensions needed for BulkCompiler. For comparison, we also model a plain, non-chunk-based multiprocessor.

We model a multicore with 4 single-issue processors running at 4 GHz. Each processor has a 4-way, 64-Kbyte L1 data cache with 64-byte lines. If the cache overflows while executing an atomic region, the chunk gets squashed. Given that the processor model is simple, we report performance in number of cycles taken by the program assuming a constant CPI of 1, irrespective of the instruction type, or whether an access hits or misses in the cache. In some of the experiments, we will assign a fixed cost in cycles to each CAS (Compare-And-Swap) operation. CAS is used to implement synchronization in the Hotspot JVM. In all cases, the results are measured after the application has run a sufficient number of instructions to warm up the code cache.

5.2 Experiments and Applications

We start by identifying which synchronization variables in the application have high contention and which have low contention. For this, we use Hotspot, which provides options to profile dynamic locking behavior. It is as simple as running with an additional Hotspot argument. This information enables the targeting of the atomic regions. In addition, our infrastructure uses a simple model of the data footprint of each code section, which is used to decide when the atomic region should terminate, to minimize cache overflow. We often chop loops into multiple blocks of appropriate sizes in order to put each block inside an atomic region.

For the evaluation, we use the SPECJBB2005 and SPECJVM98 benchmark suites. In addition, we also evaluate two additional applications with substantial synchronization, namely *MonteCarlo* from SPECJVM2008 and *JLex* from [2]. Of these applications, SPECJBB2005 and *MonteCarlo* run with 4 threads, and *Mtrt* of SPECJVM98 runs with 2 threads. The rest of SPECJVM98 and *JLex* run with a single thread, although they have many synchronized blocks. These synchronizations are in the Java library code, which includes synchronization because it has to be thread safe. Each application runs for at least 1B instructions before being measured.

Finally, among the SPECJVM98 applications, we could not evaluate *Javac* or *MpegAudio* because they are commercial applications with no source code, which we need for source level instrumentation. However, we were able to include *Jack* (another SPECJVM98 commercial application) because it has become open source under the name of JavaCC. The JavaCC source distribution includes an input set which is an identical copy of the input set for *Jack* with a few syntactic modifications.

6. EVALUATION

In this evaluation, we first describe the optimizations that we enable, then present the simulated speedups, and finally characterize the transformations performed.

6.1 Understanding the Optimizations Enabled

To understand the way in which BulkCompiler's transformations enable Hotspot to generate faster code, we analyzed the intermediate representation of the code generated by Hotspot with and without the BulkCompilerchanges. We did not add any new compiler optimization to take advantage of chunk-based execution; conventional Hotspot optimizations perform significantly better once Hotspot is given control of the chunks. The following are some common patterns seen:

Loop unswitching. This transformation involves moving a loop-invariant test out of a loop, and then producing two versions of the loop, one in the if-branch of the test, and the other in the else-branch. With the removal of the test, the two loop bodies have a more streamlined control flow and, therefore, the compiler can optimize them, creating better-quality code. The presence of synchronization within the loop had prevented this optimization, since it would have been in violation of the Java Memory Model. However, after BulkCompiler has wrapped the loop inside an atomic region and replaced the synchronizations with plain accesses, Hotspot performs this optimization automatically. The Java Memory Model will not be violated because the hardware guarantees that there are

no intervening conflicting accesses until the atomic region runs to completion.

Null check elimination. In order to satisfy Java safety guarantees, the compiler needs to insert null checks before every object reference — unless it is able to prove that the reference is non-null. If the compiler can prove that two references point to the same object, it can safely remove the checks on the second reference. This situation occurs often inside a loop, where a reference remains invariant through all the iterations. In this case, the compiler peels off the first iteration of the loop, where it inserts all the checks, and removes the checks from the main body of the loop. Hotspot could not do this optimization if there were intervening synchronizations between the references, since it would be illegal. After BulkCompiler's transformations, Hotspot performs this optimization.

Range check elimination. In addition to performing null checks, the compiler is also required to check that an array reference does not exceed the boundaries of the array. Like for null checks, if the compiler is able to prove that an earlier range check subsumes a later range check, the later check can be removed. Once again, however, the presence of intervening synchronizations prevented Hotspot to perform the same loop-peeling optimization in the code described above. With BulkCompiler's transformations, Hotspot performs the optimization.

Loop invariant code motion. Often, the same expression is computed at every iteration of a loop. A common example is when the range of an array which does not change in size needs to be computed repeatedly within a loop. This transformation involves moving the computation outside the loop. If the loop has synchronizations, Hotspot cannot move the computation. With BulkCompiler's transformations, Hotspot can perform the optimization without violating the Java or SC memory models.

Register allocation. Memory locations that were allocated in registers cannot survive synchronization boundaries. The data needs to be stored to memory and loaded back from it, or the Java Memory Model would be violated. BulkCompiler's transformations result in the removal of many register allocation restrictions, which often result in much more efficient code.

Besides these types of optimizations, the removal of memory fences done by BulkCompiler gives Hotspot much more room for code scheduling. Scheduling is especially important for potentially long delay loads and stores. However, this effect is not evaluated in our results due to the simplistic timing model used in our simulator.

6.2 Simulated Speedups

To estimate the performance gains enabled by BulkCompiler, we simulate two environments. The first one (*Baseline*) is unmodified Java running on a conventional (i.e., without chunks) multiprocessor. The second one (*BulkCompiler*) is code transformed by BulkCompiler running on a BulkSC multiprocessor.

As indicated before, because of the model used in our simulator, we report performance in number of cycles taken by the programs assuming a constant CPI of 1, irrespective of the instruction type. For this reason, we call the two environments above *Baseline_1* and *BulkCompiler_1*. However, it is well known that an important source of overhead in implementations of Java is the actual read-modify-write operations (e.g., CAS) performed in the frequent synchronizations — in the case of Hotspot, potentially two read-modify-write operations for each synchronized block, one at the beginning and one the end. BulkCompiler's transformations replace these operations with plain accesses. Consequently, in our simulations, we also report results for a second scenario, namely one where each instruction takes 1 cycle except for the read-modify-write operations, which take 20 cycles each. The latter is the overhead measured in our workstations for a read-modify-write operation. We call the two environments *Baseline_20* and *BulkCompiler_20* for the two architectures.

Since these environments do not include a high-fidelity architectural model, they do not capture how different memory models use microarchitectures for access overlapping. However, they capture how the compiler can re-order and transform the code under different models, changing the number of instructions executed.

Figure 5(a) shows, for each application, the speedup of *BulkCompiler_1* over *Baseline_1*, while Figure 5(b) shows the speedup of *BulkCompiler_20* over *Baseline_20*. The bars also include the average for the SPECJVM98 applications, and the average for all the applications. Recall that *BulkCompiler* delivers SC execution, while *Baseline* executes with the relaxed Java Memory Model.

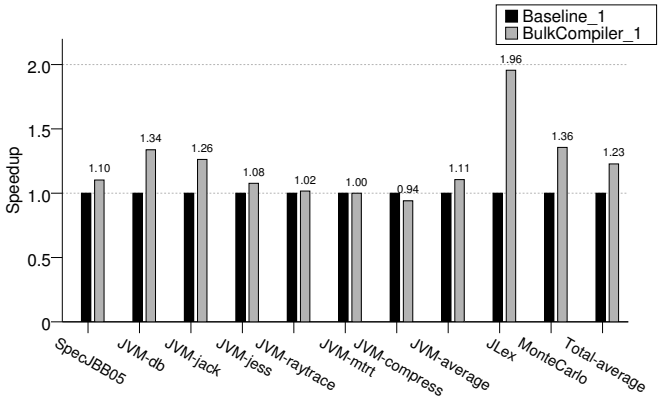
The figures show that *BulkCompiler* delivers substantial speedups over *Baseline*. In the environment where all the instructions have the same cost, the average speedup of *BulkCompiler_1* across all the applications is 1.23 (or 1.11 if we only consider SPECJVM98). In the environment where the read-modify-write instructions are more costly, which we consider to be more realistic, the speedups are higher. Specifically, the average speedup is 1.37 (or 1.23 if we only consider SPECJVM98). These results show that a *whole-system SC platform*, which guarantees SC at both the compiler and hardware levels, can deliver higher performance than a state-of-the art platform that supports the relaxed Java Memory Model (*Baseline*).

An analysis of the applications shows that most of them get speedups, sometimes quite high. The exceptions are *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress*. We did not get speedups for these applications largely because they do not contain much synchronization in the first place. However, also notice that instrumenting with atomic regions and enforcing SC did not cause them to slow down significantly, either. This is despite the fact that we wrap the *BulkCompiler* assembly instructions in JNI calls (Section 5.1), which introduce some overhead. Such overhead would not be present in an implementation that works on the Hotspot intermediate representation. Finally, we note that the speedups of *JLex* are the same for *BulkCompiler_1* and *BulkCompiler_20*. This is because the locks in *JLex* were mostly in the biased [25] state, which does not use any read-modify-write operations.

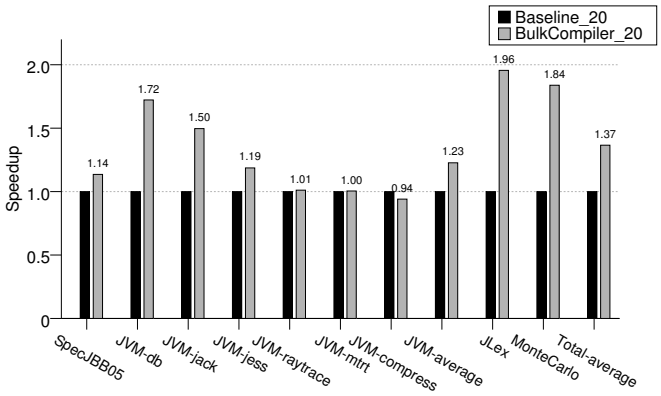
6.3 Characterizing the Transformations

In this section, we characterize the dynamic behavior of the code transformed by BulkCompiler as it runs on the BulkSC architecture. The data is shown in Table 3, where AR stands for Atomic Region. In the table, Columns 2–4 show the percentage of dynamic instructions inside atomic regions in the program, the number of dynamic atomic regions, and the average dynamic size of an atomic region in instructions, respectively. We can see from this data that our atomic regions cover the great majority of the execution (74% of the dynamic instructions on average). The remaining execution largely contains private references. We also see that there are many dynamic atomic regions and that they are very large — about 52,000 dynamic instructions on average. These atomic regions are largely loops with small to modest write footprints. The average atomic region size for *JVM-compress* is smaller than the others. This is because system calls interspersed across this application force the creation of smaller atomic regions. At this size, the overhead of our JNI calls becomes more significant and, hence, we suffer a 6% overhead as can be seen in Figure 5, even with a negligible squash rate.

Columns 5–7 give more information about these atomic regions, namely the number of synchronized blocks per region in the origi-



(a)



(b)

Figure 5: Speedups of *BulkCompiler_1* over *Baseline_1* (a), and of *BulkCompiler_20* over *Baseline_20* (b).

nal code, and their write and read footprints in number of 64-byte lines, respectively. We can see that, on average, each atomic region used to contain about 600 synchronized blocks. By transforming their synchronization operations into plain memory accesses, we enable many optimizations in Hotspot. As mentioned in Section 6.2, *JVM-raytrace*, *JVM-mtrt*, and *JVM-compress* do not have much synchronization and, therefore, show no speedups.

We also see that the atomic regions have a small write footprint (184 lines on average). This allows them to fit inside the cache without overflows. The read footprint is larger, but recall that, in BulkSC the read footprint *does not need to remain in the cache* — signatures keep a record of the lines read [6].

For example, *JVM-db* has a large read footprint but a tiny write footprint compared to the size of its atomic regions. This is due to the fact that *JVM-db* spends the bulk of its time sorting its database index, which involves string comparisons of index entries and swaps when entries are out of order. The index is only updated on swaps, which are much less frequent than the number of read accesses required for the string comparisons. This is the reason for the small write footprint. However, each access to the index is protected by a synchronized block, giving BulkCompiler ample optimization opportunities. Other applications follow a similar pattern.

Finally, the last column shows the fraction of dynamic instructions in atomic regions that get squashed. We see that, on average, only 0.48% of the instructions in atomic regions get squashed. This represents a tolerable fraction of work lost.

7. DISCUSSION

These experiments are only an initial estimation of the potential of exposing an architecture with all-the-time group commits to the compiler. Indeed, we need a high-fidelity model of the microarchitecture to assess whether BulkCompiler's higher freedom to schedule long-latency memory accesses within a large atomic region translates into performance impact.

Moreover, this paper has focused only on (i) synchronization-related issues and (ii) enabling *conventional* compiler optimizations that already exist in Hotspot — such as register allocation or loop-invariant code motion. BulkCompiler can be augmented with *novel* compiler optimizations enabled by the all-the-time group-commit hardware. Some of these optimizations could focus on speeding-up single-thread execution — an area explored by Neelakantam *et al* [21] (Section 8). Other optimizations could specifically focus on other multithreaded issues such as load imbalance.

Another avenue of research is to apply the memory ordering relaxation provided by all-the-time group commits to improve the performance of other memory consistency models beyond SC. We

Application	% of Dyn Instructions in ARs	# of Dynamic ARs	Dyn AR Size	# Sync Blocks per AR	Write Footprint (Lines)	Read Footprint (Lines)	% Instructions in AR Squashed
SPECJBB05	44.5	323086	19117.2	212	489.4	865.6	0.79
JVM-db	75.8	22451	119176.0	2000	84.4	3123.0	0.40
JVM-jack	29.5	2382	30105.2	792	119.7	229.4	1.31
JVM-jess	62.6	33995	43475.6	102	141.1	449.7	0.27
JVM-raytrace	85.8	61419	19771.1	0	51.7	613.9	0.10
JVM-mtrt	77.5	61627	19589.0	0	305.5	1297.0	0.14
JVM-compress	92.7	1632082	5418.6	0	28.1	144.5	0.04
JLex	97.4	45846	131474.0	317	426.9	705.7	0.91
MonteCarlo	99.9	16778	82535.1	2000	11.0	13.0	0.34
Average	74.0	244407	52295.8	602	184.2	826.9	0.48

Table 3: Characterizing the dynamic behavior of the code transformed by BulkCompiler. AR stands for Atomic Region.

are confident that our techniques can improve the performance of relaxed memory models as well. Work by Wenisch *et al* [32] and Blundell *et al* [3] point to the potential of these ideas.

Finally, this work is applicable beyond BulkSC to all all-the-time group-commit architectures and, with some extensions, to conventional architectures that support hardware TM.

8. RELATED WORK

8.1 Software-Only Sequential Consistency

There have been three major software-only efforts to enforce SC in programs that are not well synchronized. The most sophisticated one is the Pensieve Project [28], which provides SC for Java. Their SC compiler uses a combination of escape analysis [28], thread-structure analysis [28], delay set analysis [26, 28], and an optimized fence-insertion algorithm [10]. All but the fence-insertion algorithm are interprocedural analyses that are fairly complex. Overall, their method induces slowdowns of over 10% on average over the relaxed Java Memory Model.

Liblit *et al* [17] developed an SC version of Titanium [13]. In the same project, Krishnamurthy and Yelick [14] showed how the regular structure of SPMD programs could be exploited to reduce the complexity of delay set analysis in those programs. Finally, Von Praun and Gross [31] used an object-based analysis for delay set analysis to determine reference orders that needed to be enforced because of inter-thread conflicts. Overall, none of these methods reported speedups for applications, and some reported significant slowdowns in one or more applications. In contrast, our combined hardware-software SC scheme delivers speedups over the relaxed Java Memory Model.

8.2 Exploiting Support for Atomicity

There has been substantial recent work on exploiting hardware support for atomicity. The Transmeta Code Morphing concept involved aggressively optimizing the code with speculative transformations [8]. It appears that most of the optimizations were for single-thread execution. Neelakantam *et al* [21] sped-up hot sections of the code by developing an optimized, speculative "trace" of the code and running it under hardware atomicity. If the code takes an unexpected control path, the section is squashed and the full version of the code is executed. They largely focus on optimizing single-thread execution, typically in loop iterations, although they mention the application of SLE to critical sections. BulkCompiler differs in its emphasis on grouping many low-contention critical sections in a large atomic region to enable conventional optimizations. It also differs in its goal to support SC.

Carlstrom *et al* [5] take lock-based Java programs and convert them into transactions. They describe how critical sections and other constructs are converted into transactions. However, they neither mention whether this change enables compiler optimizations nor are they focused on SC. Other authors such as Ziarek *et al* [33] and Rossbach *et al* [24] have studied environments that integrate locks and transactions, finding some of the problems we faced.

Re-writing a critical section with a synchronization-free fast path executing under atomic hardware, and a slow path with the complete code has been proposed in SLE [23] and used in TM libraries [9].

9. CONCLUSIONS

A platform that provides high-performance SC at the hardware and software levels for all codes, including those with data races, will substantially simplify the task of programmers. This paper presented the hardware-compiler interface, and the main ideas for *BulkCompiler*, a compiler layer that works with the BulkSC chunking hardware to provide a *whole-system high-performance SC platform*. Our specific contributions included: (i) ISA primitives for BulkCompiler to interface to the chunking hardware, (ii) compiler algorithms to drive chunking and code transformations to exploit chunks, and (iii) initial results of our algorithms on Java programs.

Our results used Java application suites modified with our compiler algorithms and compiled with Sun's Hotspot server compiler. A whole-system SC environment with BulkCompiler and simulated BulkSC hardware outperformed a simulated conventional hardware platform that used the more relaxed Java Memory Model by an average of 37%. The speedups came from code optimization inside software-assembled instruction chunks.

This work is applicable beyond BulkSC to all group-commit architectures and, with some extensions, to conventional architectures that support hardware TM. We are now extending BulkCompiler to drive novel compiler optimizations for single- and multi-threading, and to apply them to relaxed memory models as well.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the I-ACOMA group members for their comments. This work was supported in part by the National Science Foundation under grants CNS 07-20593 and CCR 03-25603; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); Sun Microsystems under the University of Illinois OpenSPARC Center of Excellence; and a gift from IBM.

11. REFERENCES

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

[2] E. Berk. JLex: A Lexical Analyzer Generator for Java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

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

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

[5] B. Carlstrom et al. Transactional Execution of Java Programs. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

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

[7] S. Chaudhry et al. Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor. In *International Symposium on Computer Architecture*, June 2009.

[8] J. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation and Optimization*, March 2003.

[9] D. Dice et al. Applications of the Adaptive Transactional Memory Test Platform. In *Workshop on Transactional Computing*, February 2008.

[10] X. Fang, J. Lee, and S. P. Midkiff. Automatic Fence Insertion for Shared Memory Multiprocessing. In *International Conference on Supercomputing*, June 2003.

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

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

[13] A. Kamil, J. Su, and K. A. Yelick. Making Sequential Consistency Practical in Titanium. In *International Conference on Supercomputing*, November 2005.

[14] A. Krishnamurthy and K. A. Yelick. Analyses and Optimizations for Shared Address Space Programs. *Journal of Parallel and Distributed Computing*, November 1996.

[15] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, September 1979.

[16] K. Lee and S. P. Midkiff. A Two-Phase Escape Analysis for Parallel Java Programs. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[17] B. Liblit, A. Aiken, and K. A. Yelick. Type Systems for Distributed Data Sharing. In *International Static Analysis Symposium*, June 2003.

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

[19] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *International Symposium on Programming Language Design and Implementation*, June 2007.

[20] G. Naumovich and G. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In *International Symposium on Foundations of Software Engineering*, November 1998.

[21] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *International Symposium on Computer Architecture*, June 2007.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotspotTM Server Compiler. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium*, April 2001.

[23] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *International Symposium on Microarchitecture*, December 2001.

[24] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Symposium on Operating Systems Principles*, October 2007.

[25] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2006.

[26] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *Transactions on Programming Languages and Systems*, April 1988.

[27] Sun Microsystems. OpenJDK. `http://openjdk.java.net/`.

[28] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Symposium on Principles and Practice of Parallel Programming*, June 2005.

[29] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. *International Conference on Pervasive Services*, July 2005.

[30] Virtutech. Simics. `http://www.simics.net/`.

[31] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *Conference on Programming Language Design and Implementation*, June 2003.

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

[33] L. Ziarek et al. A Uniform Transactional Execution Environment for Java. In *European Conference on Object-Oriented Programming*, July 2008.