HARDWARE ATOMICITY: AN EFFECTIVE ABSTRACTION FOR RELIABLE SOFTWARE SPECULATION

HARDWARE SUPPORT FOR ATOMIC EXECUTION CAN BOTH GREATLY SIMPLIFY THE IMPLEMENTATION OF EXISTING SPECULATIVE COMPILER OPTIMIZATIONS AND ENABLE NEW ONES. GIVEN CURRENT TECHNOLOGY TRENDS, THIS HARDWARE AND SOFTWARE COOPERATION IS A COMPELLING APPROACH; SUCH OPTIMIZATIONS CAN SIMULTANEOUSLY IMPROVE SINGLE-THREAD PERFORMANCE AND REDUCE POWER CONSUMPTION IN BOTH SEQUENTIAL AND MULTITHREADED APPLICATIONS.

Naveen Neelakantam
Craig Zilles
University of Illinois at Urbana-Champaign

Ravi Rajwar
Suresh Srinivas
Uma Srinivasan
Intel

Technology trends and shrinking power envelopes have forced microprocessor designers to focus on hardware techniques that efficiently improve single-thread performance without superlinear increases in power and silicon area. Although aggressive compiler optimizations offer a means to improve performance and reduce instruction overheads, their implementation complexity has greatly limited their use and impact. Hardware features that enable simple implementations of aggressive compiler optimizations and bring them into widespread use by making trivial their correct implementation can help improve both performance and power.

In this article, we identify *hardware atomic execution*—the execution of a region of code completely (and as if all operations in the region occurred at one instant) or not at all—as such a feature for simplifying existing and enabling new speculative compiler optimizations. Specifically, we propose that microprocessors expose atomic execution as a hardware primitive to the compiler. Doing so lets the compiler generate a speculative version of the code where infrequently executed code paths are removed. These code paths don't have to be considered in (and hence do not constrain) the region's optimization. If such a pruned path must execute, the hardware aborts the execution of the speculative version and restores the execution state to the beginning of the code region. Control then transfers to a nonspeculative version of the code. Although conventional approaches to implementing speculative compiler optimizations require extensive compensation code—a major source of complexity—hardware atomicity obviates the need for compensation code.

Hardware atomicity as a simplifying primitive

To demonstrate how hardware atomicity can alleviate the complexity incurred by conventional speculative compiler optimi-

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society

zations, we use a representative example. Consider the dynamic interprocedural control-flow graph (CFG) of the fully optimized code generated by a leading commercial Java virtual machine (JVM) for the most frequently executed loop from the DaCapo benchmark[1] jython (see Figure 1a). Few paths through this loop ever execute, but the hottest path executes 109 conditional branches and more than 600 instructions. Our manual analysis of the hot path found that aggressive speculative optimizations could remove more than two-thirds of the instructions (Figure 1b). Conventional implementations for speculative optimization incur significant complexity because the compiler must preserve correct execution along all potential paths through the loop. Figure 1c shows the simplest possible CFG—having aggressively eliminated 67 branches with redundant conditions—that achieves the desired hot-path optimization. Many commercial systems don't perform such aggressive speculative optimizations because they find it difficult to verify the correctness of these radical program transformations.

Figure 1c demonstrates the compiler's inability to isolate hot paths from cold paths, showing the fundamental source of complexity for the CFG. The compiler must guarantee that any exit from the hot path, however unlikely, will generate correct results. For this, the compiler must ensure two things. First, the compiler must save sufficient program state in the hot path so that, at each exit, it can construct the precise program state required by that cold path. Second, it must maintain mappings from the optimized hot path's state to that of the cold path. This lets the compiler generate compensation code for every exit from the hot path to undo any hot-path-specific optimizations.

Atomic execution, however, obviates the need for this complexity, as Figure 1d shows. First, the compiler replicates the hot code for execution in an atomic region. The compiler denotes the start of the atomic region with an instruction (aregion_begin) to communicate the beginning of speculative execution to the hardware, as well as the exits from the atomic region with an instruction (aregion_end) to instruct the

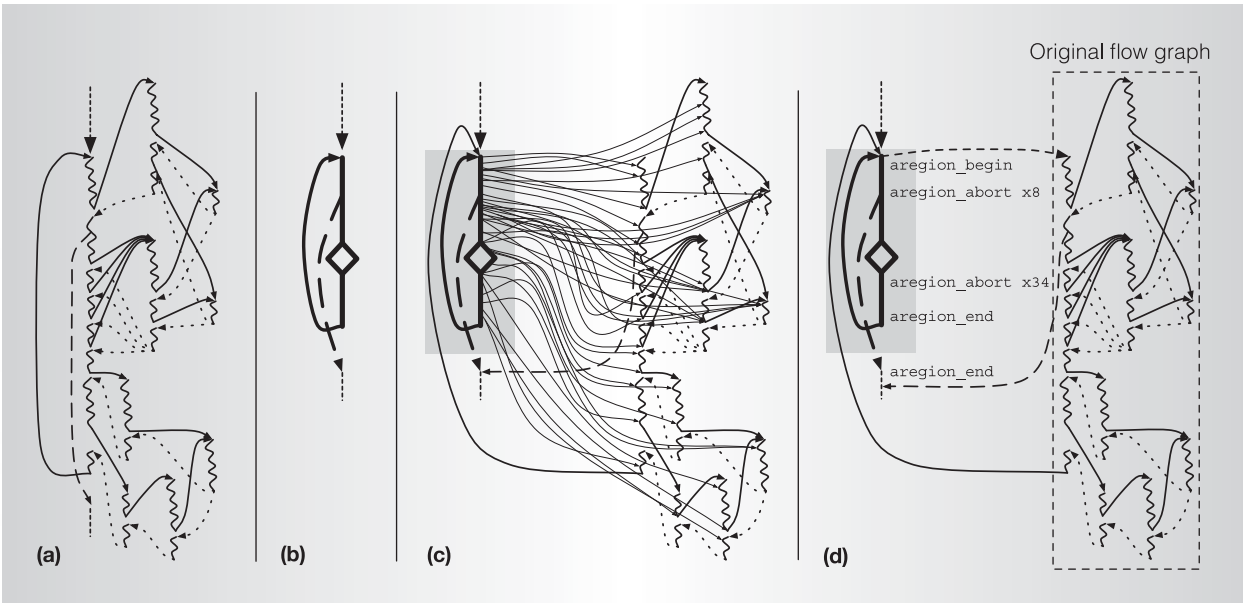

Figure 1. Dynamic interprocedural control-flow graphs (CFGs) for a hot loop with a deep call graph: as optimized (with significant inlining) by a leading Java virtual machine (a), as manually optimized (with full inlining) after removing the cold paths (b), as could be implemented with conventional speculative optimization (c), and as could be implemented with hardware atomic execution (d).

hardware to commit the region's results atomically. The compiler then converts branches to cold paths into conditional abort instructions (aregion_abort). If an abort condition evaluates such that control should transfer to a cold path, the hardware rolls back to the state prior to the aregion_begin and transfers control to the original (nonspeculative) version of the code, as if speculative execution of the hot path had never occurred. In this way, the hardware relieves the compiler from the responsibility of generating compensation code; the hot-path code merely needs to verify its speculation—for example, to verify that a cold path isn't taken.

Specifically, the hardware atomicity abstraction simplifies implementation of speculative optimizations in the following ways:

- The hardware maintains the state necessary to recover from speculative optimizations. The compiler no longer needs to generate compensation code to recover the correct program state at each exit from the hot path.
- Hardware atomicity enables analysis and optimizations to ignore the cold paths when optimizing the hot paths. By converting cold paths into conditional aborts, the compiler enables existing nonspeculative analysis routines and optimizations to perform, in effect, path-qualified analysis and speculative optimizations. Previous approaches to speculative optimization required a complete reimplementation of these compiler passes.
- The hardware isolates execution from other threads. The compiler needn't worry about the multithreaded safety of optimizations within the atomic region because memory operations in the region appear to occur atomically with respect to memory operations from other threads.

For these reasons, hardware atomicity greatly improves the return on investment for implementing compiler optimizations. By using atomic regions, we enable much higher code quality for a given amount of compiler complexity. That is, hardware atomicity improves an optimization's effectiveness, simplifies its implementation, or both.

Atomic execution primitives also provide a clean abstraction for compiler implementation. We find that incorporating atomic regions into an existing compiler intermediate representation requires minimal reengineering of existing intermediate-representation nodes and compiler passes; we simply leverage existing exception handling mechanisms. In addition, atomic regions can be formed early in the compilation process, benefiting optimizations and transformations on both the high-level intermediate representation (such as inlining and loop unrolling) and low-level intermediate representation. Previous hardware support for compiler optimizations (such as IA-64) provided overly complicated abstractions, and their usefulness was restricted to backend optimizations such as scheduling.

Not only do atomic regions enable existing nonspeculative formulations of optimizations to perform transformations that typically require speculative formulations, the atomic region abstraction also simplifies the implementation of new optimizations. For example, one of the authors of this article produced a working implementation of partial inlining in six hours. Without hardware atomic regions, this is a difficult transformation to implement. When discussing the correctness of their partial inliner implementation (which didn't use atomic regions), Muth and Debray remarked, "The flow of control in the program resulting from partial inlining is sufficiently complex that it is no longer obvious that the resulting program is semantically equivalent to the original."[2]

We find, however, that a high-performance implementation of hardware atomicity is a necessity because any overhead translates directly into lost performance.

Providing hardware atomicity

Providing hardware atomicity is a natural extension to modern processors. Modern processors employ speculative execution and typically record information at an instruction granularity to restore execution state when speculation fails. However, specula-

tion mostly succeeds, and processors frequently don't need the recorded information. Checkpoint processors use this observation to optimize recovery information management by recording recovery state at coarse intervals (hundreds of instructions).[3] We extend this checkpoint abstraction to provide hardware atomicity by ensuring that groups of memory updates appear to commit instantaneously; all other memory operations in the system appear to occur either before or after these operations.

We expose hardware atomicity to software by permitting software to delimit atomic regions. These atomic regions have the following invariant: Either the processor commits the region successfully, or it undoes all changes performed in the region and transfers control to an alternate region. The compiler attempts to execute a speculative version of the code as an atomic region, and if the execution is unsuccessful, it falls back to a less aggressively optimized version that includes all paths.

We expose three instructions to the compiler:

- aregion_begin <alternate PC> marks the start of a speculatively optimized region. It also specifies the alternate code path for aborts as an operand. To make undo possible, the hardware also creates a recovery point for the register state (for example, similar to a branch instruction creating a rename table checkpoint).
- aregion_end marks the end of the region. The hardware commits the region's memory updates atomically.
- aregion_abort permits the runtime software to explicitly undo the atomic region, such as when the execution must proceed down a path not included in the speculative version.

Because the software uses hardware atomicity to opportunistically improve a single thread's performance, the hardware implementation only has to deal with common execution scenarios. For example, the processor treats hardware limitations such as limited buffering and infrequent events such as exceptions and interrupts as implicit exits from the hot path. In practice, such simplifications have little negative impact because the compiler can select atomic regions to generally avoid them. Forward progress is guaranteed because the compiler always provides a nonspeculative implementation to handle any cases where hardware can't provide atomicity.

Two special-purpose registers, visible to software, are updated when an atomic region aborts. The first register encodes the reasons for an abort (such as explicit abort, interrupt, data conflict, and exception). The second register records the program counter of the instruction responsible for an abort, if any. This information allows a runtime system, like a JVM, to diagnose the cause of aborts and adaptively recompile to maintain a low misspeculation rate.

The simple interface provides significant flexibility to hardware designers: A hardware implementation can execute the code region in whatever way seems fit, as long as when an abort condition occurs, the execution restores to the beginning of the region with appropriate information in the appropriate registers. Our implementation builds on a checkpoint architecture substrate to achieve a nearly no-overhead execution of the common case (of no aborts) and permits multiple atomic regions to be in flight simultaneously. Various implementation strategies based on checkpoint architectures exist for hardware atomicity. We use the data cache to retain the atomic region's data footprint and a register rename table checkpoint to recover register state. The data cache extends each cache line with two bits: one to track read addresses and another to track written addresses. The processor treats external snoop invalidations to lines with a read or write bit set and external snoop reads to lines with the write bit set as abort conditions. The processor performs flash clear operations to commit or abort speculative state.

Compiler optimizations using hardware atomicity

We now describe how the compiler exploits hardware atomic regions to improve the generated code quality using an

illustrative, but representative example. (See our earlier work for a detailed description of the compiler implementation.[4])

Figure 2a shows the control flow graphs (CFGs) of two example methods that depict a common optimization idiom: method foo calls the synchronized method bar, and the compiler considers inlining bar into foo to expose optimization opportunities. The compiler has annotated the CFG edges for both methods with taken frequencies derived from an edge profile, as most runtime optimizers do prior to optimization. The monitor_enter and monitor_exit intrinsics in the method bar provide the mutual-exclusion property that the Java synchronized keyword requires. Basic block Y contains an operation that could incur an exception and therefore has an outgoing exception edge that the profile indicates has never been taken. The exception edge is connected to another monitor_exit intrinsic, which will free the synchronization lock before invoking exception dispatch.

The example depicts several common optimization obstacles. In both methods, extremely rare execution paths (C → B, X → call, and Y → exception) limit the optimization of more frequently executed paths because these infrequent paths likely use or redefine variables that obscure optimization opportunity. The static size of infrequently executed paths can also prevent the compiler from inlining methods—for example, bar into foo. The monitor_enter and monitor_exit intrinsics will expand into the relatively complex CFGs required of high-performance lock implementations.[5] Furthermore, monitor_enter and monitor_exit are synchronization actions in the Java memory model, which restrict the compiler's ability to perform optimizations across them.[6]

The atomic region abstraction provides the compiler writer with a simple, yet effective means of overcoming these common performance obstacles. It enables the compiler writer to reason about exploiting available optimization opportunity without being concerned about infrequently executed paths or multiprocessor memory models. For example, in Figure 2, the compiler replaces each infrequently executed path with an operation to assert that the path isn't followed. Similarly, the compiler replaces the balanced pair of synchronization actions with an operation that asserts that no other thread holds the lock. The all-or-nothing property of atomic regions aids both transformations; if any of the assertions computes to false, the processor discards the updates performed by the speculative optimizations, making it as if the speculative execution had never happened. Furthermore, the instantaneous commit provided by an atomic region lets the compiler remove synchronization actions safely because the hardware prevents unsafe interleaving of memory operations from other threads.[7]

Figure 2b shows the CFG for a region the compiler formed to exploit atomic execution. The region is a duplicate of the CFG in Figure 2a, where bar has been partially inlined into foo and infrequently executed paths have been removed. To ensure correctness despite the removal of some paths, the compiler replaces the branch to each removed path with an assert operation to check that the expected path was taken. By traversing the CFG for this region, the compiler can identify the balanced pair of monitor_enter and monitor_exit intrinsics and convert them into an assertion that the monitor is free. The resulting atomic region is reconnected into the flow graph for foo by pointing all of A's in-edges to the aregion_begin and adding an exception edge from the aregion_begin to A.

Figure 2c shows the atomic region formed and the complete CFG for a speculatively optimized foo. The simplified CFG contained within the atomic region enables the compiler to transform and schedule the common program paths without having to generate compensation code.

Importantly, the compiler uses the existing exception handling mechanisms in the intermediate representation to represent the atomic region in the optimized CFG of Figure 2c. To the compiler, an atomic region abort appears as if an exception occurred in the aregion_begin block and transferred control to block A. By inserting the exception edge between these two

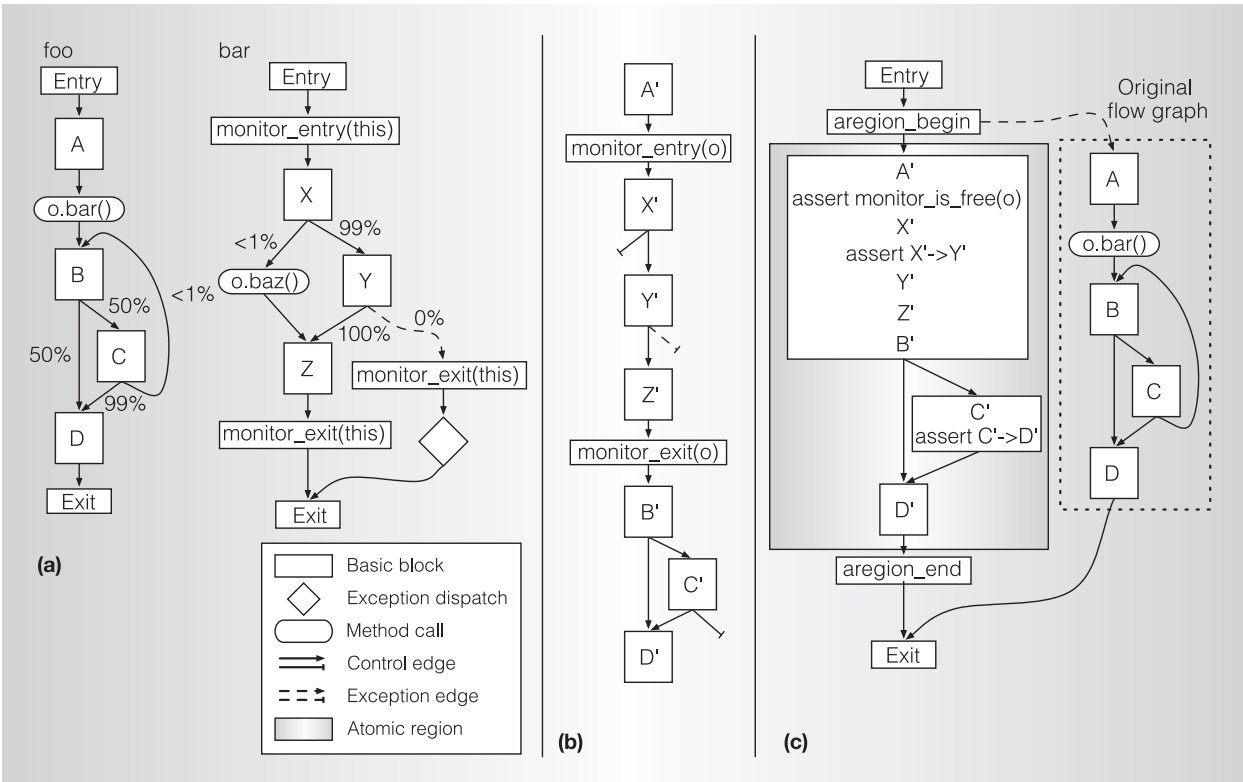

Figure 2. Atomic region formation by the compiler: initial control flow graphs (CFGs) for the methods annotated with a control flow profile (a), a replicated version of the hot paths after partial inlining and trimming of cold paths (b), and the final CFG for the method foo (c).

blocks, the compiler preserves the values needed by the abort path and performs register allocation appropriately. The assert operations are a simple addition to the intermediate representation and are represented as (noncontrol) arithmetic operations that have source values but produce no output. As a result, existing compiler passes can eliminate redundant assertions and schedule them.

By using existing compiler mechanisms to represent atomic regions, we don't have to modify any existing optimization passes to exploit the exposed speculative optimization opportunity.

Key results

We evaluated atomic-region-based speculative optimizations using the Apache Harmony Dynamic Runtime Layer Virtual Machine (DRLVM) JVM (http://harmony.apache.org/subcomponents/drlvm). We modified the JVM to support atomic region formation, partial loop unrolling, partial inlining, and software-controlled speculative lock elision (SLE). We used the DRLVM's server execution manager configuration to maximize the baseline code quality. To confirm that the benefits we achieve aren't simply due to scope enlargement from the partial inlining, we used a baseline with an unrealistically large inlining threshold (five times larger than the server's default) to mitigate the benefits of further inlining. The compilation process is completely automatic and profile driven. For our simulation environment, we used an x86 full-system simulator coupled with a detailed timing model of a conventional out-of-order processor extended to support atomic regions. We simulated benchmarks from the DaCapo suite[1] using the sampling method we describe in the "Efficiently Evaluating Codesigned Hardware-Software Techniques in a Managed-Runtime Context" sidebar.

Figure 3 shows the benefits of atomic regions, both in terms of performance and dynamic micro-operation (μop) count reduction. We report the reduction in dynamic μop counts, because they generally translate into improved energy efficiency; fewer μops flowing down the pipeline will result in less switching activity, which in turn results in reduced energy consumed to perform a given unit of program work.

These optimizations provide an average of 17 percent improvement in performance across our benchmarks with an almost commensurate reduction (15 percent average) in the number of μops retired. A strong correlation between μop reduction and speedup isn't surprising, because both generally result from more effective optimization. Our optimizations don't simply remove instructions, they also replace operations with simpler alternatives and simplify and reduce the critical path through the code. For example, SLE replaces compare-and-swap primitives and monitor-data-structure updates with a load and a branch. This is also why speedups often exceed μop reductions.

Two factors largely determine the variation in results between benchmarks: coverage and abort rate. Coverage is the dominant factor. Table 1 shows that four of the benchmarks with high speedups—bloat, hsqldb, jython, and xalan—execute most (upwards of 69 percent) of their μops in atomic regions. As we report coverage after optimization and most of the reduction in dynamic μop count occurs in the atomic regions, atomic regions encapsulate an even larger fraction of the program than these coverage numbers suggest. The antlr benchmark achieves a good speedup in spite of low coverage because it optimizes away more than half of the instructions in the atomic regions it does form.

Three benchmarks incur abort rates exceeding 1 percent (see Table 1). These aborts are due to the program's profile changing during its execution—paths that initially appear cold becoming frequently executed later in the execution. Adaptive recompilation can address such changing profiles.[8] Eliminating the small number of atomic regions with high abort rates

Efficiently Evaluating Codesigned Hardware-Software Techniques in a Managed-Runtime Context

As always, to perform a fair, apples-to-apples performance comparison, we need to compare equivalent regions of a program's execution. However, this can be particularly challenging to perform for hardware-software codesigned techniques in a managed runtime environment, like the Java virtual machine (JVM), for three reasons:

- Compilation and optimization is performed during the program run; as a result, the benchmark runs must be sufficiently long for the staged optimizer to produce the fully optimized code.
- The overhead of detailed timing simulation precludes using full benchmark runs in timing simulation. Sampling is a necessity.
- The generated code will be different for the runs with and without the proposed technique, precluding sampling approaches that specify simulation regions by their instruction offsets—for example, by sampling approaches that skip 2.7 billion instructions, then simulate for 100 million.

To achieve a fair comparison within these constraints, we developed a method whereby markers are inserted into the executable to delineate the boundaries of the region to simulate. Specifically, we modify the JVM's compiler to insert special marker instructions in the function prologue of a user-specified method. The user also specifies a 3-tuple $<N, M, O>$, which indicates that the program should be functionally simulated up to the marker's Nth occurrence, caches and branch predictors should be warmed up until the Mth marker, and detailed timing simulation should be performed from the Mth to the Oth marker. The simulator treats the markers as nops, but it counts their occurrences and performs the necessary simulator mode conversions.

To select good marker locations, we collect a complete trace of method invocations from the benchmark's execution. We break this trace into blocks of 10,000 methods and use SimPoint 3.0's phase classification tool[1] to identify phases. For up to four phases per benchmark, we automatically select a marker method that is infrequently invoked (so that it minimally perturbs the execution) and can be used to bound a representative simulation sample. This method has some similarities to concurrent work.[2]

This technique is sufficiently general that we've implemented it in both Intel's SoftSDV and Virtutech's Simics full-system simulators. We find that full-system simulation is a necessity because most JVMs and many of their workloads are multithreaded and use many system features. Although the nondeterminism introduced by multithreading can prevent our marker-based technique from being able to record equivalent work in two runs, we find that, if most of the application execution is performed in one thread, we can reliably capture equivalent execution regions late in the program's execution when little JVM work (such as compilation) is being performed. Only in DaCapo benchmarks lusearch and xalan did nondeterminism present problems; we used an older, single-threaded version of xalan (beta-2006-08) so that we could include it in our results.

References

1. G. Hamerly et al., ''SimPoint 3.0: Faster and More Flexible Program Analysis,'' *J. Instruction Level Parallelism*, vol. 7, Sept. 2005, pp. 1-28; www.jilp.org/vol7/v7paper14.pdf.
2. E. Perelman et al., ''Cross Binary Simulation Points,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS), IEEE CS Press, 2007, pp. 179-189.

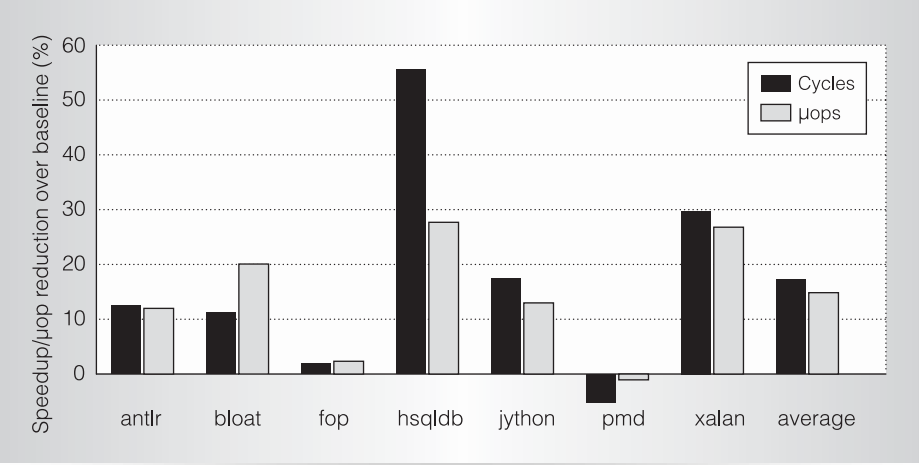

Figure 3. Execution time speedup and micro-operation (µop) count reductions due to improved compilation.

eliminates pmd's slowdown and brings bloat's speedup inline with its µop reduction.

Significant opportunities remain to improve coverage and region selection as well as to increase the aggressiveness of optimization in the context of an adaptive system. We view our compiler as only an initial prototype.

Microarchitecture implications of atomic regions

We find that, for the benchmarks we studied and our compiler implementation, most atomic regions exceed typical pipeline resources but often fit in the data cache. Most regions access fewer than 10 cache lines, and 50 cache lines provide sufficient buffering for 99 percent of the atomic regions. Moreover, region selection can tolerate the hardware constraints of an atomic primitive implementation. For example, with our region selection algorithm, of the nearly 1.7 million atomic regions executed, only one region overflowed the hardware resources.

Hardware implementations of atomic regions might appear similar in function to hardware support for transactional memory,[9] but key differences in requirements and usage result in different hardware implementation requirements. Atomic re-

Table 1. Atomic region statistics.

	Atomic regions			Region abort rate	
Benchmark	**Coverage (%)***	**Unique****	**Size*****	**%†**	**per 1,000 µops††**
antlr	9	96	47	0.02	0.0004
bloat	69	93	128	4.30	0.12
fop	20	73	32	0.01	0.0007
hsqldb	76	75	88	2.74	0.24
jython	87	14	227	0.69	0.27
pmd	32	32	42	2.20	0.18
xalan	78	37	78	0.28	0.03

* Fraction of µops executed in atomic regions.
** Average number of unique atomic regions in execution sample(s).
*** Average size of atomic regions (in dynamic instructions).
† Percentage of regions aborting.
†† Number of aborts per 1,000 µops.

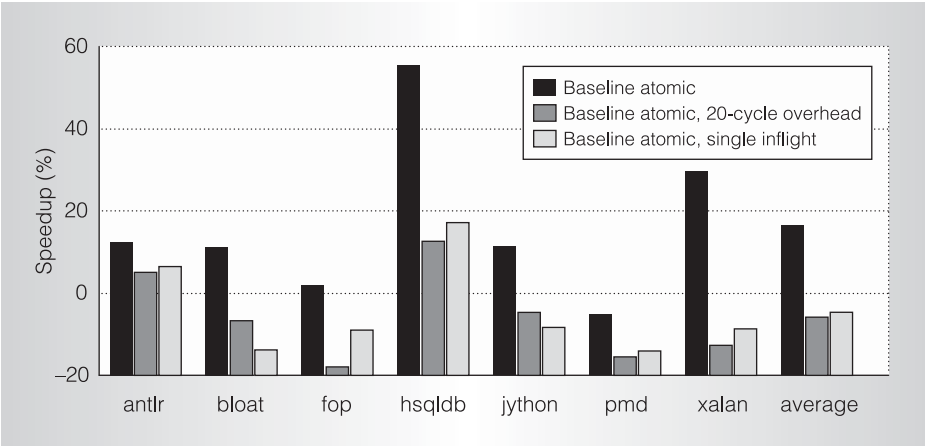

Figure 4. Sensitivity to implementation simplifications of the hardware atomic primitive (performance normalized to compilation without atomic regions).

gions require high-performance hardware implementations because these regions focus on improving single-thread performance, and any execution overhead reduces the benefits. In contrast, transactional memory (proposed for scalability) might be able to tolerate some loss in a single thread's performance if the resulting scaling sufficiently overcomes it.

Up to this point, our results assume a high-performance implementation of hardware atomicity. We can also consider two potential sources of overhead that might be present in simplified implementations: overhead in the form of additional operations and serialization that might occur as part of the aregion_begin to record recovery state and the inability to have more than one atomic region in flight at a time. We performed two experiments to measure the sensitivity to these effects: configuring the simulator to stall the pipeline for 20 cycles when processing an aregion_begin, and stalling an aregion_begin at decode until any preceding atomic region commits.

Figure 4 compares these hardware configurations with the baseline checkpoint-based implementation (all using the same software configuration), showing that either simplification significantly impacts performance, causing most applications to yield slowdowns. Only antlr shows limited sensitivity, because it uses atomic regions rather sparingly and gets significant benefits from the ones it employs.

Related work in optimization

Exposing hardware checkpoint and rollback support to the compiler was first proposed as part of the block-structured ISA.[10] Although a compiler was never built for the block-structured ISA, it did introduce several important concepts which would later form the basis of rePLay: the assert instruction and the atomic block. The assert instruction is similar to the assertions in our compiler's intermediate representation, and the atomic block is a more restricted form of our atomic region (the atomic block is a single-entry single-exit block, whereas the atomic region is single-entry, multiple-exit and can contain internal control-flow). The block-structured ISA differs from our proposal in that it exposes atomicity via a radically different and complicated instruction-set design whereas our proposal extends an existing instruction set architecture (ISA) with a simple and concise set of primitives.

Prior work on code generation for architectures with hardware checkpoint and rollback support has been exclusively in the context of dynamic binary optimizers, whereas our work demonstrates its integration into traditional compilation.

The rePLay framework converts predictable control flow into assertions to create atomic blocks for optimization underneath the ISA level.[11] It uses dedicated hardware to implement profiling, region formation, and

optimization, thereby limiting flexibility and increasing complexity. The optimized regions aren't persistent and cause programs with large instruction footprints to unnecessarily reconstruct and reoptimize previously optimized regions.

The Transmeta Crusoe processor also provides checkpoint and rollback capability for compiler optimizations.[12] The Transmeta Code Morphing Software (CMS) uses this capability to implement precise architectural exceptions and for recovery when loads incorrectly schedule ahead of stores. Unlike our proposal, it doesn't use this capability for explicit control speculation—that is, it has no explicit abort.

The dynamic binary optimization approach of prior work constrains them to perform optimizations at the ISA level. Our approach permits higher-level optimizations such as speculative lock elision (SLE) and partial inlining. Our use of hardware to implement atomic execution lets us perform aggressive optimizations and still be compatible with strict memory consistency models. Our regions are also less constrained. rePLay precludes conditional branches in optimized regions and limits region size to that of the reorder buffer. Crusoe's implementation limits the region size to that of the gated store buffer and the memory disambiguation hardware. Our proposal doesn't constrain the type of control flow in the region and tolerates regions that can reside in the data cache.

In his landmark paper ''Compilers and Computer Architecture,'' William Wulf identifies three principles (regularity, orthogonality, and composability) that instruction sets should adhere to in order to simplify compiler implementations, thereby improving the code quality that is practically achievable.[13] Each time instructions-set designers don't observe these principles, an additional set of special cases must be considered during compilation to generate the best possible code for a given program. Although architectures that ignore these principles do not, in theory, preclude the building of compilers that generate the highest performance code, in practice the quality of code suffers as many compiler implementations will be unable to justify the additional software complexity required.

In the spirit of the principles set forth by Wulf, we see atomicity as a fundamental abstraction that greatly simplifies the implementation of speculative compiler optimizations. By exposing a primitive that enables a region of code to be executed completely and instantaneously or not at all, we permit the compiler to separate the concerns of generating optimized code for the common case and generating correct code for all possible cases.

Looking forward, we believe that this simplifying atomic abstraction will enable researchers to further push the envelope on how aggressive speculative optimizations can be. In addition, the ability of atomic regions to prune cold paths from consideration will enable compilers to greatly expand their optimization scope with little of the code bloat that's traditionally associated with inlining-based scope expansion. MICRO

Acknowledgments

This research was supported in part by the US National Science Foundation CCR-0311340, NSF CAREER award CCR-03047260, and a gift from Intel.

References

1. S.M. Blackburn et al., ''The DaCapo Benchmarks: Java Benchmarking Development and Analysis,'' *Proc. 21st Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 06), ACM Press, 2006, pp. 169-190.
2. R. Muth and S. Debray, ''Partial Inlining,'' tech. report, Dept. of Computer Science, Univ. of Arizona, 1997.
3. H. Akkary, R. Rajwar, and S.T. Srinivasan, ''Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,'' *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 03), IEEE CS Press, 2003, pp. 423-434.
4. N. Neelakantam et al., ''Hardware Atomicity for Reliable Software Speculation,'' *Proc. 34th Ann. Int'l Symp. Computer*

Architecture (ISCA 07), IEEE CS Press, 2007, pp. 174-185.
5. K. Kawachiya, A. Koseki, and T. Onodera, ''Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations,'' *Proc. 17th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 02), ACM Press, 2002, pp. 130-141.
6. J. Manson, W. Pugh, and S.V. Adve, ''The Java Memory Model,'' *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages* (POPL 05), ACM Press, 2005, pp. 378-391.
7. R. Rajwar and J.R. Goodman, ''Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution,'' *Proc. 34th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 01), IEEE CS Press, 2001, pp. 294-305.
8. C. Zilles and N. Neelakantam, ''Reactive Techniques for Controlling Software Speculation,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO 05), ACM Press, 2005, pp. 305-316.
9. J.R. Larus and R. Rajwar, *Transactional Memory*, Morgan and Claypool, 2006.
10. S. Melvin and Y. Patt, ''Enhancing Instruction Scheduling with a Block-Structured ISA,'' *Int'l J. Parallel Programming*, vol. 23, no. 3, 1995, pp. 221-243.
11. S.J. Patel and S.S. Lumetta, ''rePLay: A Hardware Framework for Dynamic Optimization,'' *IEEE Trans. Computers*, vol. 50, no. 6, 2001, pp. 590-608.
12. J.C. Dehnert et al., ''The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges,'' *Proc. Int'l Symp. Code Generation and Optimization* (CGO), ACM Press, 2003, pp. 15-24.
13. W.A. Wulf, ''Compilers and Computer Architecture,'' *Computer*, vol. 14, no. 7, 1981, pp. 41-47.

Naveen Neelakantam is a PhD student in computer science at the University of Illinois at Urbana-Champaign and is an Intel Foundation PhD Fellowship recipient. His research interests include microprocessor architecture and optimizing compilers. Neelakantam has an MS in electrical engineering from the University of Illinois at Urbana-Champaign. He is a student member of the ACM.

Ravi Rajwar is an architect in the Digital Enterprise Group at Intel. His research interests include the theoretical and practical aspects of computer architecture. Rajwar has a PhD in computer science from the University of Wisconsin–Madison.

Suresh Srinivas is a principal engineer in the Software and Solutions Group within Intel, where he currently focuses on new technology development using open-source technologies. Srinivas has a PhD in computer science from Indiana University.

Uma Srinivasan is a senior staff software engineer in the Software and Solutions Group and is the technical lead for the Java on Itanium Just-in-Time Compiler project in the Java/XML Engineering Lab at Intel. Srinivasan has an MS in computer science from the University of Wisconsin–Madison.

Craig Zilles is an assistant professor in computer science at the University of Illinois at Urbana-Champaign. His research interests include interaction between compilers and computer architecture, especially in the context of managed and dynamic languages. Zilles has a PhD in computer science from the University of Wisconsin–Madison. He is a member of IEEE and the ACM.

Direct questions and comments about this article to Naveen Neelakantam, Siebel Center for Computer Science, M/C 258, 201 N. Goodwin Ave., Urbana, IL 61801; neelakan@uiuc.edu.