A Real System Evaluation of Hardware Atomicity for Software Speculation

Naveen Neelakantam†‡, David R. Ditzel‡, and Craig Zilles†
University of Illinois at Urbana-Champaign† and Intel Corporation‡
[neelakan, zilles]@illinois.edu, david.r.ditzel@intel.com

Abstract

In this paper we evaluate the atomic region *compiler abstraction by incorporating it into a commercial system. We find that atomic regions are simple and intuitive to integrate into an x86 binary-translation system. Furthermore, doing so trivially enables additional optimization opportunities beyond that achievable by a high-performance dynamic optimizer, which already implements superblocks.*

We show that atomic regions can suffer from severe performance penalties if misspeculations are left uncontrolled, but that a simple software control mechanism is sufficient to reign in all detrimental side-effects. We evaluate using full reference runs of the SPEC CPU2000 integer benchmarks and find that atomic regions enable up to a 9% (3% on average) improvement beyond the performance of a tuned product.

These performance improvements are achieved without any negative side effects. Performance side effects such as code bloat are absent with atomic regions; in fact, static code size is reduced. The hardware necessary is synergistic with other needs and was already available on the commercial product used in our evaluation. Finally, the software complexity is minimal as a single developer was able to incorporate atomic regions into a sophisticated 300,000 line code base in three months, despite never having seen the translator source code beforehand.

Categories and Subject Descriptors: D.3.4 [**Software**]: Programming Languages–*Processors:Compilers, Optimization*, C.0 [**Computer Systems Organization**]: General–*Hardware/software interfaces*

Keywords: Atomicity, Checkpoint, Optimization, Speculation, Dynamic Translation

General Terms: Performance

1. Introduction

After decades of favorable semiconductor scaling, hardware designers now face a much more challenging regime characterized by limited voltage scaling, severe power and thermal limits, and wire dominated delays. In response, hardware proposals for increasing single-thread performance face a strict standard: performance improvements must be provided without undue increase in power and complexity. For example, designers of the Intel®Pentium®M microprocessor only accepted microarchitectural proposals that cost less than 3% in additional power for every 1% of performance improvement[7]. In the power and complexity conscious environment of today, the constraints are even more severe.

New compiler optimizations offer an opportunity to both improve the performance of a single thread as well as reduce power consumption, but developing optimizations that surpass the state-of-the-art typically involves complex implementations that are time consuming and difficult to develop. Therefore, cost-effective hardware that can aid a compiler in achieving new heights of performance are of particular interest.

One such feature is *hardware atomicity*: the execution of a region of code either completely or not at all[14]. Using the hardware atomicity primitive, a compiler can easily construct *atomic regions* that trivially expose speculative opportunities to classical optimizations. These atomic regions enable a compiler to optimize common program paths by simply removing rarely-taken paths. Should a removed path be needed, hardware discards all speculative updates, rolls back the atomic region and then resumes execution at a non-speculatively optimized version of the same code.

Our previous work has already demonstrated that atomic regions, by facilitating improved code generation in a JVM, can improve the performance of Java* programs by 10% on average. However, our work relied on short simulations using a modified JVM and therefore suffered from several experimental shortcomings. For example, we could not measure extra runtime compilation costs that might overwhelm the benefits achieved. Likewise, an atomic region optimization may become unprofitable as a program changes phases and complete program runs are necessary to fully observe and properly react to such effects. Finally, our evaluation and implementation focused on managed languages and ignored pre-compiled native binaries.

Our previous work also presented atomic regions as a simple and effective alternative to other more complicated abstractions such as the superblock. A compiler can use the atomic region abstraction to isolate frequently executed program paths from cold paths, which enables unmodified optimizations to trivially exploit speculative opportunities. In contrast, the superblock requires new compiler optimizations to identify and exploit speculative opportunity and often relies on complex compensation code to guarantee correctness if a misspeculation occurs.

However, our previous evaluation used as its baseline an optimizing compiler which makes no use of superblocks. It left open the question of whether or not atomic regions offer utility in an infrastructure that already makes heavy use of superblocks. Likewise, the atomic region abstraction presumes the existence of atomic ex-

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.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03. . . $10.00

* Other names and brands may be claimed as the property of others.

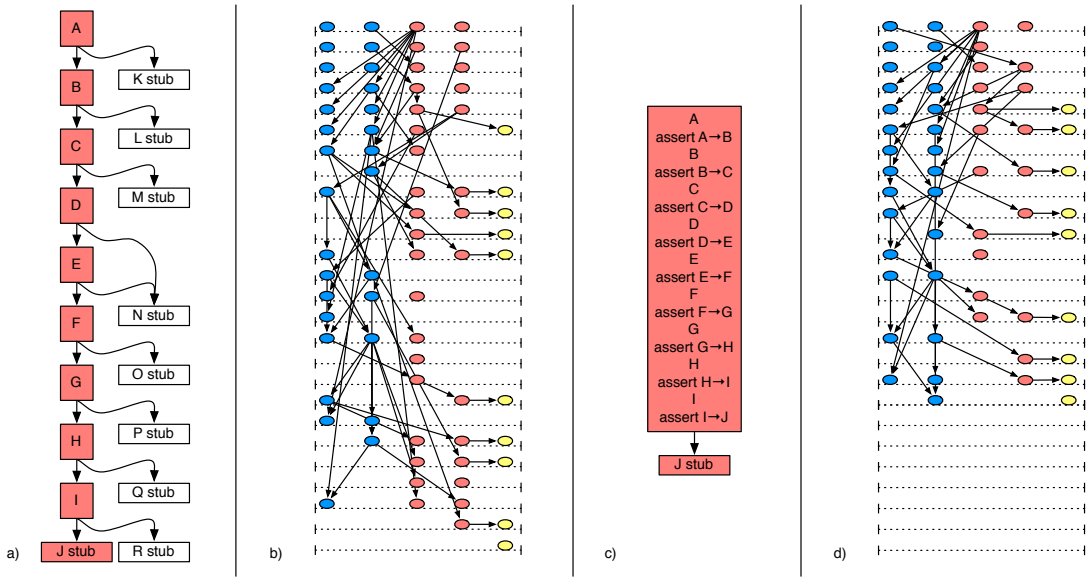

Figure 1. Potential for atomic region optimizations. Optimization region for method OaGetObject from *vortex*. (a) control flow graph for the optimization region with its hot path highlighted, (b) dataflow graph and schedule for the region as optimized by CMS using superblocks, (c) atomic region representation of the same control flow graph after cold paths have been converted into asserts, (d) dataflow graph and schedule after the atomic region has been optimized. Each node in the dataflow graphs depicts an Efficeon operation: the two left columns depict memory operations, the next two columns depict integer operations and the rightmost column depicts branches. Each arrow in the dataflow graph depicts a true dependence. Note that branch conditions can be consumed the same cycle they are produced.

ecution hardware and therefore a fair discussion requires providing similar hardware features for superblocks.

To address these shortcomings, this work integrates atomic regions into a binary translator composed of co-designed hardware and software. Specifically, we use a Transmeta* development system which includes a late-model Transmeta Efficeon* processor, source code to the Transmeta Code Morphing Software*(CMS), and the hardware necessary to flash a modified CMS back onto the system. The Efficeon processor already includes precisely the support advocated for atomic regions, namely fast hardware checkpoint and rollback primitives (Section 3).

Using this hardware platform, we make the following contributions:

- **We demonstrate that atomic regions are easy to integrate into a mature, high-performance compilation system.** The Transmeta Code Morphing Software is a dynamic translation product capable of high-performance execution of x86 code on dissimilar hardware. It has been meticulously engineered and has undergone a decade of dedicated development. Incorporating a new compilation strategy into such a system is ordinarily a major undertaking. Nevertheless, the simplicity of the atomic region abstraction facilitated its incorporation in just three months by a single developer—despite never having seen any of the 300,000 lines of CMS source code beforehand. We elaborate upon this point in Section 4.
- **We argue that atomic regions complement the use of superblocks.** We find that while the support for atomic regions does improve the performance of CMS, which makes heavy use of superblocks[1], atomic regions are a complement rather than a substitute for superblocks.
- **We develop a simple and effective mechanism for reacting to profile changes.** With atomic regions, recovering from a misspeculation is easy but costly. Recovery requires hardware rollback, which discards work, and redirecting execution to an alternate, often low-performance, implementation of the same code. Therefore, identifying unprofitable speculations and disabling them is critical to performance. In Section 4.3 we describe a simple and sufficient mechanism for doing so.
- **We evaluate atomic regions in a real hardware system.** Using real hardware, we are able to measure the effects of atomic regions on complete SPEC CPU2000 runs. Our results include all system effects including compilation overhead and misspeculations triggered by profile changes (Section 5).

We continue in Section 2 with a simple example to help illustrate how atomic regions are used and how they compare to optimization strategies used in high-performance compiler infrastructures.

2. Illustrative Comparison

In this section, we show how the atomic region abstraction benefits a mature compilation system, even one designed around the superblock. We use an example from the SPEC CPU2000 benchmark *255.vortex* to illustrate our findings.

Shown in Figure 1(a) is the control flow graph (CFG) for a portion of the method `OaGetObject`. The portion shown is the optimization region selected by the Transmeta CMS translator. This is one of the hottest regions in *vortex* and accounts for 7% of the overall execution time. A single hot path exists through this optimization region, including 56 x86 instruction and nine cold exits.

CMS uses superblock formation in conjunction with a suite of classical optimizations to generate the Efficeon code and schedule shown in Figure 1(b). This code has been aggressively scheduled

[1] CMS actually uses hyperblocks, which for this discussion we consider a subset of superblocks.

and optimized including the speculative hoisting of loads, reordering of memory operations and removal of redundant operations. As described in previous work[5], this requires no compensation code because CMS makes use of the atomic execution hardware provided by Efficeon.

Ignoring cache misses, this optimized code emulates the original 56 x86 instructions by executing 72 Efficeon operations in 26 cycles or an average of 2.15 x86 instructions per cycle (IPC) and 2.77 IPC in Efficeon operations. The code generated by CMS is of high quality: the static schedule produced has a similar height to the dynamic schedule achieved by a modern out-of-order processor given the same number of functional units.

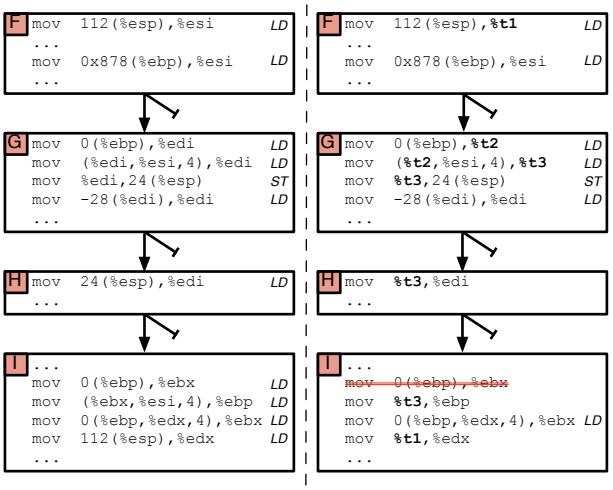

Figure 2. CMS baseline optimization. CMS uses temporaries to eliminate three redundant loads and eliminate a fourth load by forwarding a previously stored value. Original x86 code is shown on the left. The code on the right illustrates the effect of optimized CMS code.

Figure 2 shows a few of the optimizations that CMS applies to the region from Figure 1(a). In particular, CMS is able to identify three redundant load operations and eliminate them by buffering previously loaded values into temporary registers. Likewise, CMS forwards a value stored to the stack to a later consumer, which obviates a fourth load. In each of these cases, CMS uses liveness analysis to prevent unnecessary register copying (for example, the temporaries introduced in block **G** are never copied to register `edi` because of a subsequent kill).

Although the code generated by CMS is already of high quality, hand inspection shows that significant optimization opportunity remains. Put simply, superblock scheduling and optimization has enabled CMS to eliminate some redundancies, hoist critical operations past exits and generate a nearly optimal schedule, but it has not enabled CMS to remove operations that are only needed along cold exit paths.

Because the code generated by CMS already uses hardware primitives to execute the region atomically, these additional optimization opportunities can be trivially exposed by converting the cold exits into *asserts*, as advocated by the atomic region abstraction (shown in Figure 1(c)).

An assert operation simply performs a conditional check to verify that a cold exit has not been followed. If the cold exit is followed, the assert triggers an abort which causes the entire atomic region to be rolled back and redirects execution to code which includes the cold exit. An assert therefore enables the compiler to speculatively isolate frequently occurring paths in the CFG from rarely taken exits.

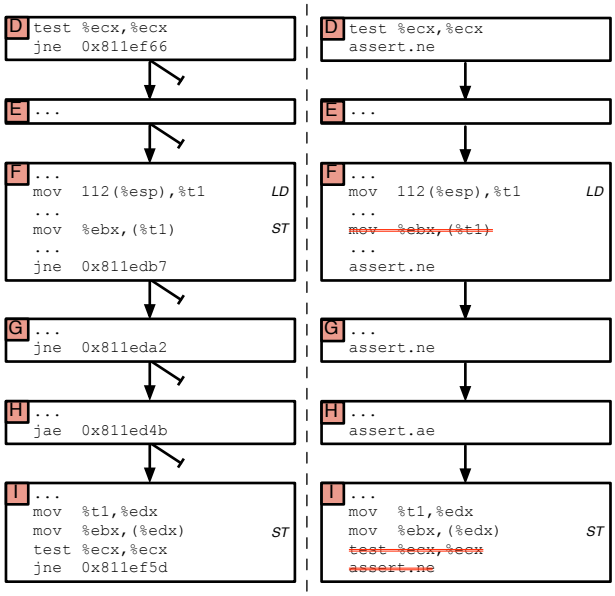

Figure 3. Optimizations enabled by atomic regions. Asserts trivially expose speculative opportunities to classical optimizations. For example, a partially dead store gains the appearance of a fully dead store and a partially redundant exit computation appears fully redundant.

Figure 3 demonstrates a few of the additional opportunities the atomic region abstraction exposes in the same region from Figure 1(a). By converting cold exits into simple dataflow operations, an assert provides speculative opportunities with a non-speculative appearance. For example, after converting the cold exits in blocks **F**, **G**, and **H** the partially dead store in block **F** appears fully redundant to classical optimizations in CMS and is thereby eliminated. In addition, the partially redundant branch exit computation in block **D** becomes fully redundant after assert conversion (enabling the removal of the redundancies in block **I**).

Figure 1(d) shows the optimized code and schedule that results after converting cold exits into asserts. By removing the cold exits, CMS is able to remove 15 additional Efficeon operations from the region. Given fewer operations and fewer control-flow constraints, the superblock scheduler generates a 27% shorter schedule. This more aggressively optimized code now executes in 19 cycles at an average x86 IPC of 2.95.

This example region clearly shows the performance potential offered by the atomic region abstraction[2]. As we will show, the benefits demonstrated in this example also translate into performance gains for full programs running on a real system (for example, we show a 9% performance improvement on reference runs of *vortex*). We will further show that a simple control mechanism is all that is necessary to avoid detrimental performance side effects.

3. Background

The Transmeta Efficeon, first released in 2003, utilizes a low complexity design to provide high-performance and low-power x86

[2] The example also implies that a compiler must choose the same optimization scope for atomic regions as it would for superblocks. However, atomic regions are more general than superblocks because they can contain arbitrary control flow and therefore can encapsulate larger optimization scopes. While there could be a benefit to taking advantage of this difference, we do not explore it in this paper.

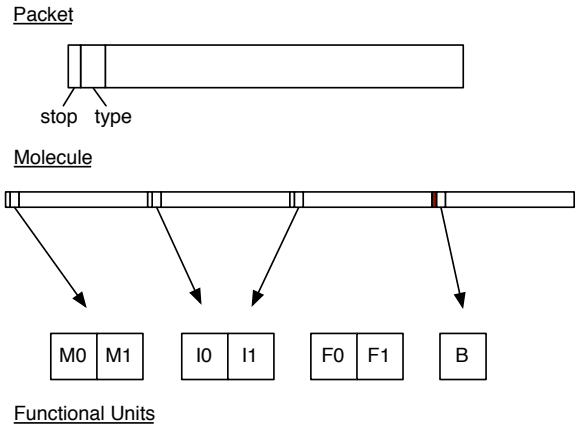

Figure 4. The Efficeon architecture. The Efficeon executes molecules composed of a variable number of 32-bit packets. Each packet includes a *stop* bit as well as a *type* field that statically encodes which functional unit it uses.

compatibility. The Efficeon hardware is a very long instruction word (VLIW) processor that executes an entirely different instruction set than x86. The instruction set is designed to enable the Code Morphing Software (CMS) software system to faithfully execute x86 code through interpretation and by dynamically translating x86 code into high-performance native code.

In Section 3.1, we briefly describe the architecture of the Efficeon processor. A brief overview of CMS and how it is used to provide high-performance execution of x86 binaries on the Efficeon follows in Section 3.2.

3.1 Efficeon Processor Architecture

The Efficeon is an in-order VLIW processor, which is designed to provide high-frequency execution of software-scheduled code. To further simplify the design and reduce power, it does not provide hardware interlocks or register scoreboarding and therefore relies completely upon a compiler to correctly schedule dependent and independent operations. To simplify the compiler's task, Efficeon provides hardware support for taking fast register and memory checkpoints and for reordering memory operations.

Depicted in Figure 4, an Efficeon VLIW instruction, or *molecule*, is variable length and composed of 32-bit *packets*. Each packet includes a *stop* bit, which denotes the end of a molecule. A molecule may contain up to eight packets.

A packet typically encodes a functional operation, or *atom*, but may also encode auxiliary information such as longer immediates or memory alias protection. An Efficeon atom has a three-address format and is analogous to an instruction from a load-store architecture. An atom is statically assigned to one of seven functional units: two memory, two integer, two floating point and one branch.

The Efficeon processor provides hardware support for fast register and memory checkpoints. The Efficeon has two copies of each register: a *shadowed* and a *working* copy. Likewise, the Efficeon includes a *speculative* bit for each line in its data cache[18]. Between checkpoints, all updates are speculatively written to either working registers or the data cache. If a cache line is speculatively written, its speculative bit is set and it is transitioned to the dirty state (after first evicting any non-speculative dirty data on the line into a victim cache). The hardware can *commit* speculative work in a single cycle by copying the working registers onto their shadowed counterparts and flash clearing all speculative bits in the data cache. Alternatively, the hardware can *rollback* all speculative work by restoring the working registers from their shadowed counterparts and flash invalidating all speculative lines in the data cache. Section 4.1 describes the primitives used by software to control this commit and rollback hardware.

The Efficeon also provides memory alias detection hardware, which a compiler can use to guarantee the correctness of reordered memory operations. Often memory operations do not alias, but the compiler can not statically prove their independence. In these situations, the compiler can generate code which includes *alias* packets. If used to initiate protection of a coupled load or store atom, an alias packet captures the memory address used by the atom into an alias register. If used to detect an alias with a coupled load or store atom, the alias packet compares the memory address against the contents of one or more alias registers, and, if a match is made, an alias fault is triggered. In this way, software can check if speculatively-reordered loads alias with the stores they were hoisted above. The alias hardware also enables the compiler to eliminate redundant loads and stores in more situations[10].

The Efficeon shares many of the above architectural traits with the Transmeta Crusoe* that preceded it, but several key differences exist[10]. Both processors have statically scheduled VLIW architectures but the Efficeon can issue seven atoms per cycle versus four atoms in the Crusoe. To provide better code density, an Efficeon molecule is variable in length whereas a Crusoe molecule may only be two or four packets long. Most relevant to this paper, the Efficeon has more relaxed speculation support because it buffers all speculative memory updates in a 64-KB first level data cache whereas the Crusoe buffers all speculative memory updates in a gated store buffer.

3.2 CMS Overview

The Transmeta Code Morphing Software[5] is a software system designed to provide high-performance execution of x86 binaries on Efficeon hardware. To accomplish this goal, it includes a robust and high-performance dynamic binary translator, supported by a software x86 interpreter. The translator is a large and well-tuned software system which includes components to identify commonly-executed regions of x86 code, convert the corresponding x86 instructions into a three-address intermediate representation (IR), and then optimize, schedule, and deploy each translated region.

The first several times a group of x86 instructions is encountered by CMS they will not be translated. Rather, they will be emulated by the CMS interpreter. In doing so, CMS is able to collect a dynamic execution profile of the x86 instructions as well as provide a low-latency "cold start" response. If a group of x86 instructions is executed enough times, CMS will generate a translation for them.

A CMS translation is a multiple-entry, multiple-exit region of x86 instructions that can contain arbitrary control flow such as indirect branches, divergences, and loops. As with other dynamic translation systems, exits from a CMS translation are directly linked, or *chained*, to other translations[1, 4]. In CMS, chaining is lazily performed the first time an exit is executed.

The CMS translator uses a staged optimization strategy in managing its translations. Translations are first lightly-optimized, but later *promoted* to an aggressively-optimized translation if executed frequently enough. This staged optimization strategy enables CMS to focus its compilation efforts on the small subset of an x86 program's instructions where the majority of execution time is spent.

In this paper, we focus our efforts on improving the quality of these *aggressive* translations, and the remainder of this section focuses on the design of the aggressive optimizer.

The aggressive optimizer is designed to enable compiler optimizations to exploit as much available opportunity as possible, while keeping the total compilation time to a minimum. It primarily accomplishes these goals through a careful organization of compilation steps as described below:

1. **Region preparation:** Decode the selected x86 region into a three-address code intermediate representation (IR).
2. **Flow analysis:** Generate a topological ordering of basic blocks in the region, compute dominators and post-dominators, and rename operands into a static single-assignment (SSA) form.
3. **Control flow manipulation:** Unroll loops, if-convert short branch-overs and control-flow divergences. Also create single-entry multiple-exit sub-regions (hyperblocks) that are wrapped with checkpoint commit points to provide atomicity. Incrementally update the already computed flow analysis as necessary.
4. **Forward dataflow pass:** In a single forward pass, apply a suite of optimizations such as constant folding and propagation, common subexpression elimination, and several peephole optimizations. Also performs a simple alias analysis to guide redundant load elimination and later memory optimization and scheduling passes.
5. **Backward dataflow pass:** In a single backward pass, perform a liveness analysis to guide dead-code elimination and dead-store elimination.
6. **Schedule and lower:** Perform loop-invariant code motion, hoist critical operations, allocate registers, perform code lowering, and schedule each hyperblock.
7. **Emit:** Assemble all instructions and update branch targets.

There are two key differences between this organization and that typically employed by a static compiler. First, the CMS translator is broken into distinct phases which constrain the types of changes that can be made to the IR at any given point. For example, modifications to the control flow graph are only performed in Step 3, meaning that later passes can rely on an immutable control-flow structure. Likewise, flow analysis is performed early in Step 2 and is properly updated by the control-flow manipulation passes so that later phases can rely on accurate information about loop structure, dominators, and post-dominators.

Second, dataflow optimizations that are typically implemented as separate passes in a static compiler are instead folded into a single forward dataflow pass and a single backward dataflow pass. For example, the forward dataflow pass processes the region in topological order and applies a suite of global analysis and optimizations as it visits each statement in a basic block. In doing so, the benefits of several (in the case of CMS, seven) forward dataflow passes can be achieved in roughly the same amount of time as a single forward pass.

These design differences are key to the efficiency of the translator and, thereby, the performance of CMS as a whole. In adding additional optimizations to CMS, it is extremely important to respect these efficiency considerations. In the context of a dynamic optimizer, a powerful but computationally complex optimization is untenable. As we show next, incorporating the atomic region abstraction is not only easy to do, but can be done without adding significant overheads.

4. Atomic Regions in CMS

In this section we describe the modifications we made to CMS in order to incorporate the atomic region abstraction. We first discuss the hardware atomicity primitives that Efficeon provides and how they can be used by a software compiler to implement the atomic region abstraction. We then describe how atomic regions were integrated into CMS. Lastly, we introduce a simple mechanism for reining in frequent misspeculations and an optimization for removing redundant asserts.

`commit`	Copy *working* registers into *shadowed* registers. Mark *speculative* lines in the data cache as dirty.
`rollback`	Copy *shadowed* registers into *working* registers. Invalidate *speculative* lines in the data cache.

Table 1. Efficeon atomicity primitives. Software uses these operations to control the Efficeon commit and rollback hardware

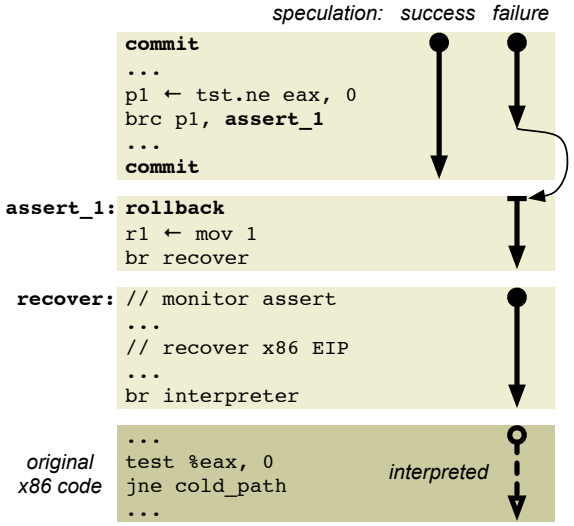

Figure 5. Atomic region example using the Efficeon atomicity primitives. If speculation succeeds, the assert path will not be taken and execution will reach the `commit` at the end of the region. If speculation fails, the abort path executes a `rollback` before invoking recovery code to restart execution at a non-speculative version of the same code (*e.g.*, via the CMS interpreter).

4.1 Hardware Atomicity

The Efficeon processor exposes its support for fast hardware checkpoints through the two operations shown in Table 1. Software can use these operations to provide the illusion of *atomic execution*—the execution of a region of code completely or not at all.

The `commit` operation is used to denote both the beginning and the end of an atomic execution region (*atomic region*). It is used at the beginning of an atomic region to take a register checkpoint and to treat all future register and memory updates as speculative. It is used at the end of an atomic region to commit all speculative updates and discard the last checkpoint. The `rollback` operation is used to unconditionally abort an atomic region by restoring the last checkpoint. A rollback does not affect the program counter, so an instruction following the rollback can be used to redirect control flow as necessary.

Figure 5 illustrates how the CMS translator can use these operations to speculatively optimize a region of code. The optimizer first wraps an optimization region with commit points, and then speculatively removes cold paths from the region. To guarantee correctness, the optimizer inserts a check, called an *assert*[15], to verify that the cold path is not taken. If the assert determines that the cold path is needed, a rollback is executed that instructs the hardware to discard all speculative state. Control is then redirected to a non-speculative version of the same region. In this example, execution resumes in the CMS interpreter.

It should be noted that the Efficeon hardware is designed to provide atomicity in a uniprocessor environment. In a multipro-

cessor environment, additional support is necessary to provide an illusion of atomicity to other threads. Essentially, loads must also be handled speculatively and coherence traffic must be monitored to detect atomicity violations. The necessary support has previously been proposed[2, 14, 17].

4.2 Incorporating Atomic Regions into CMS

The CMS translator already uses the hardware atomicity primitives to obviate the need for recovery code in superblocks. The CMS optimizer wraps each superblock with commit points to simplify the recovery of precise state in the case of a misspeculation or exception. For example, if a speculatively hoisted load incurs a memory fault, CMS relies on the hardware to discard all speculative state and afterwards redirects execution to a more conservative implementation of the same code (often by dispatching to the interpreter).

However, the CMS translator does not use hardware atomicity to expose speculative optimization opportunities resulting from biased control flow. As described in Section 2, the translator can be made to better optimize code by simply generating an atomic region with rarely executed paths removed. Extending the CMS translator to use atomic region optimizations required three additions: representing an assert operation in the IR, a mechanism for converting biased branches into asserts, and a mechanism for recovering from misspeculations.

Assert operations: We implemented the assert operation in the compiler IR as a pseudo operation. The assert is used to speculatively convert highly-biased conditional branches into straight-line code. The assert consumes the same condition as a biased branch and—like the branch it replaces—has no dataflow consumers. Unlike a branch, no operations are control dependent on an assert, which means that it is not an optimization obstacle for later passes. The assert is treated as a potentially-exception-causing operation in the IR to prevent it from being hoisted out of its atomic region, and an assert is annotated with a numerical identifier to distinguish it from other asserts in the same region.

Converting biased branches into asserts: An accurate execution profile is necessary to identify which conditional branches are good candidates to convert into asserts. Misspeculations can be very costly, so only highly-biased branches should be converted. However, CMS does not collect a profile that is sufficient to properly distinguish good candidate branches. The execution profile collected by the CMS interpreter simply does not include enough samples to be useful for our purposes.

Rather than forcing the interpreter to collect more samples or adding instrumentation code to lightly-optimized translations, both of which could incur costly performance overheads, we instead turned our attention to the translation chaining mechanism.

As described in Section 3.2, translation exits are lazily chained to other translations. Therefore when a lightly-optimized translation is promoted and retranslated, rarely taken translation exits are unlikely to have been chained. Similarly, the conditional branches corresponding to these unchained exits are likely to be biased. We have implemented a heuristic based on this observation that strikes a reasonable balance between being able to identify good assert candidates and minimizing profiling overheads.

We modified the CMS translator so that it consults chaining information when promoting a lightly-optimized translation. All unchained exits are considered assert candidates, and this information is provided to a new flow manipulation pass which we added to Step 3 of the optimizer. Figures 1(a) and 1(c) from our illustrative example show how unchained exits are converted to asserts in the aggressively-optimized control flow graph.

Misspeculation recovery: Throughout most of the optimizer, the assert is represented as a single dataflow operation. In the final code emit step this changes, and the assert is emitted as a conditional branch which targets rollback code. The exact rollback routine that the assert targets depends on the numerical identifier of the assert. Each identifier is associated with a separate rollback routine to simplify misspeculation monitoring (described in Section 4.3).

There are a maximum of 31 chainable exits in a translation, and the numerical identifier assigned to an assert is the same as the exit number of the cold branch it replaces. We therefore added 31 rollback routines to CMS which are shared by all asserts. As shown in Figure 5, a conditional branch is emitted for each assert that targets the rollback routine with the corresponding numerical identifier.

When an assert fires, control is directed to its rollback routine, which first executes a `rollback` to discard all speculative register and memory state. It then loads the identifier of the triggered assert into a register and jumps to the misspeculation recovery routine. This misspeculation recovery routine is responsible for recovering the x86 instruction pointer and dispatching to the interpreter (to execute the same code non-speculatively). The misspeculation recovery routine is also responsible for monitoring each assert, which we describe next.

4.3 Monitoring Speculations

Even though our heuristic for identifying biased branches is reasonably accurate, it is still fallible. It occasionally leads CMS to convert branches into asserts that fire too frequently. Often the cause is a change in program behavior: a path that was rarely executed early in the program becomes a common path later in the program. If these problematic asserts are left untended, they will adversely affect performance because of the relatively high cost associated with misspeculation. Therefore, a mechanism is necessary to identify and disable problematic asserts[20].

We developed a simple solution by augmenting the misspeculation recovery routine. The routine updates a misspeculation counter corresponding to the assert that fired[3]. If this counter exceeds a threshold, then the assert is designated *misbehaving*, and the translation will be reoptimized with the corresponding assert disabled.

Furthermore, it is desirable to tolerate asserts that fire infrequently relative to the total number of times they execute. For these asserts, the performance improvements provided by each successful execution of the assert outweighs the infrequent misspeculation costs. To distinguish between asserts which are problematic and asserts that are tolerable, it is ideal to know the *local assert rate*, or the number of times an assert fires relative to the number of times it executes.

Discovering the precise execution frequency of an assert is difficult, as it would require intrusive profiling. In the interest of minimizing overheads, we use an alternative approach based on hardware sampling.

By default, CMS takes a program counter sample every 200,000 cycles so that it can identify and promote frequently executing translations. If a sample is taken while a translation is executing, a counter in the translation metadata is incremented. We can therefore use this counter as an approximation for translation execution frequency.

Our assert monitoring mechanism is shown in Algorithm 1. Essentially, whenever an assert misspeculation counter is updated we also capture the value of the translation sample counter. When the next misspeculation occurs, the code checks whether a sample has been received since the last time the assert fired—by comparing

[3] The counter does not increase the size of the metadata associated with a translation because an assert replaces what would have otherwise been a translation exit. Each translation already includes eight bytes of metadata for each translation exit and we simply reappropriate the same space to house metadata for each assert.

Algorithm 1 Assert misspeculation monitoring

```
// Monitors the behavior of a misspeculating assert.
// Returns true if the assert should be disabled.
procedure MonitorAssert(assertID, transID)
    currSample ← GetTransSampleValue(transID)
    lastSample ← GetCapturedSampleValue(assertID)
    GlobalAssertCount ← GlobalAssertCount +1
    if AssertSampleMatches(currSample, lastSample) then
        assertCount ← GetAssertCount(assertID) +1
        if assertCount > AssertThresh then
            return true
        else
            SetAssertCount(assertID, assertCount)
            return false
    else
        SetAssertCount(assertID, 1)
        SetCapturedSampleValue(assertID, currSample)
        return false

// Compares two sample values. Returns true if they are equivalent
// after shifting off some of their least significant bits.
procedure AssertSampleMatches(currSample, lastSample)
    mismatchBits ← currSample ⊕ lastSample
    mismatchBits ← mismatchBits ≫ AssertSampleShift
    return mismatchBits ≡ 0

// Global assert monitoring. If the global assert rate is too high,
// tighten the local sample shift or threshold. Otherwise, loosen them.
procedure GlobalMonitor
    if GlobalAssertCount > GlobalAssertThresh then
        if AssertSampleShift < MaxSampleShift then
            AssertSampleShift ← AssertSampleShift +1
        else if AssertThresh > 0 then
            AssertThresh ← AssertThresh −1
    else
        if AssertThresh < LocalAssertThresh then
            AssertThresh ← AssertThresh +1
        else if AssertSampleShift > 0 then
            AssertSampleShift ← AssertSampleShift −1
    GlobalAssertCount ← 0
```

the captured sample value to the current translation sample counter value. A changed sample counter value implies that the translation is commonly executed, and by proxy so is the assert being monitored. To reflect that misspeculations from commonly executed asserts should be tolerated, the misspeculation counter is reset if the sample values do not match. Otherwise, when the assert count exceeds a threshold it is disabled through retranslation.

However, workloads with a large number of commonly executed translations will have an increased latency to detect misbehaving asserts. To prevent this increased detection latency from adversely affecting performance, we also incorporate a mechanism to monitor the *global assert rate*, or the total number of asserts firing per cycle.

Algorithm 1 also shows our global monitoring mechanism. Every 100 million cycles the global monitor is invoked to check if the global assert count exceeds a global assert threshold. If the threshold is exceeded, the parameters of the local assert monitoring mechanism are tightened: either by increasing the assert sample shift parameter (to require sample counter values to differ in more significant bits before considering a translation commonly executed) or by reducing the local assert threshold.

4.4 Eliminating Redundant Asserts

After biased branches have been converted into asserts, opportunities exist to remove some of these asserts from the CFG. These opportunities arise either because an assert is redundant—another assert in the same atomic region implements the same (or stronger) check—or because an assert can be proven to never fire.

The existing common subexpression elimination and constant evaluation optimizations were easily modified to recognize assert operations. We also added an optimization that can eliminate an assert if it is rendered unnecessary by a stronger assert. One assert is considered *stronger* than another if it would fire in at least every situation that the other would fire. For example, an assert that fires whenever $r1 < 5$ is stronger than an assert which fires whenever $r1 < 4$.

We also extended common subexpression elimination to allow an assert to be removed if it is *post-dominated* by an equivalent or stronger assert. Typically an operation must be *dominated* by an equivalent (or stronger) operation to be removed, but atomicity makes post-dominance a sufficient proxy for dominance.

Processor	Transmeta Efficeon 2 (TM8800)
Processor frequency	1.2 GHz
Dynamic Translator	CMS 7.0 (pre-release)
Registers	64 integer, 64 FP
Translation Cache	32 MB (of physical memory)
L1 Instruction Cache	128 KB, 4-way, 64B line
L1 Data Cache	64 KB, 8-way, 32B line
Victim Cache	1 KB, fully-associative, 32B line
L2 Unified Cache	1024 KB, 4-way, 128B line
Physical Memory	1 GB DDR-400
Operating System	Linux 2.6.19
Compiler (for SPEC)	Intel®C++ Compiler 11.0
Compilation options	-O3 -ipo -no-prec-div -prof_use
Local Assert Threshold	8 per translation sample
Global Assert Threshold	16 per 100 million cycles

Table 2. Evaluation system configuration

5. Evaluation

In this section, we present the result of incorporating the atomic region abstraction into CMS, evaluated on an Efficeon hardware platform. We first describe the configuration of our evaluation system and provide compilation details for the benchmarks used. We then present and interpret our experimental results. Overall, we find that incorporating atomic regions into CMS provides a 3% average performance improvement and that our simple assert monitoring mechanism is sufficient enough to prevent slowdowns in any individual benchmark.

5.1 System Configuration

All of our experiments are run using a Transmeta development system. Shown in Table 2, the system configuration is intended to closely represent retail Efficeon hardware[4]. Of particular note is the pre-release version of CMS that we used for our evaluation; this CMS version includes significant enhancements over the last retail version of CMS and is in many ways superior. In terms of raw performance, the pre-release version is marginally faster than the retail version on the benchmarks we used.

For our evaluation, we run all of the SPEC CPU2000 integer benchmarks to completion using the reference inputs. We did not use SPEC CPU2006 because our evaluation system—representative of a computer circa 2004—does not satisfy the system requirements. The benchmarks are compiled with the Intel®C++ Compiler using the highest performing SPEC "base"

[4] Transmeta stopped selling microprocessors in 2005.

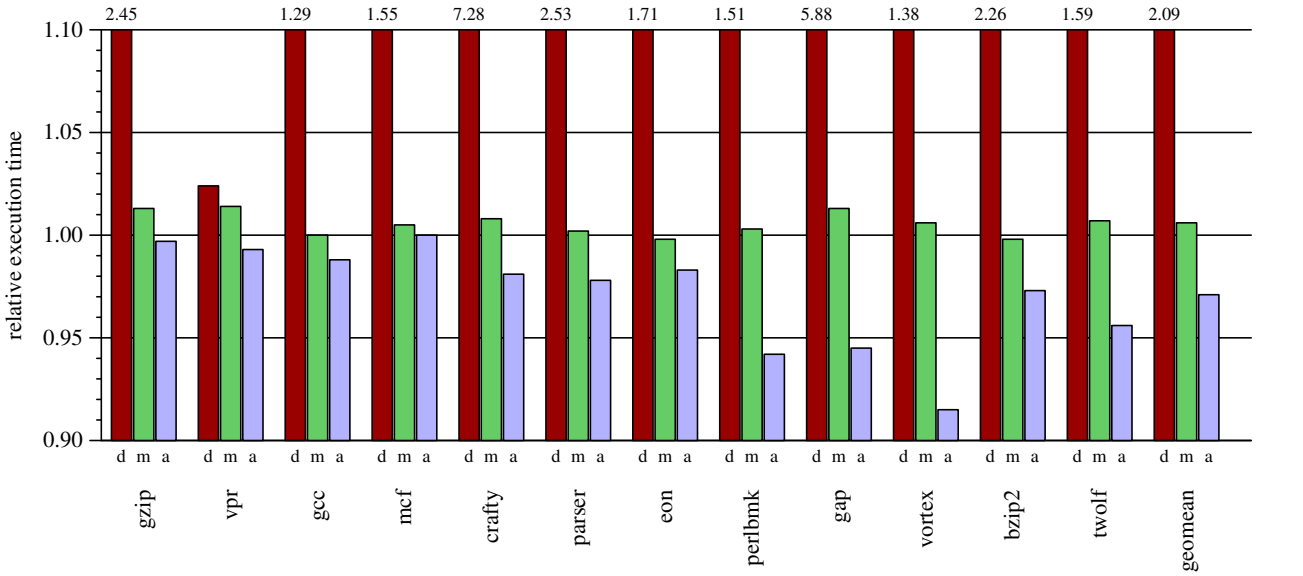

Figure 6. SPEC CPU2000 integer results. Results for three atomic region configurations. (d)isabled misspeculation monitoring, (m)onitoring enabled but assert optimizations disabled, (a)tomic region optimizations and monitoring fully enabled. All results have been normalized to the baseline CMS configuration, which has no atomic region support.

compiler options, including profile guided optimizations. All efforts have been made to use the best possible baseline.

5.2 Experimental Results

In our experiments, we focus on understanding both the dynamic and static impacts of atomic regions. We find that atomic regions are able to improve performance over a baseline CMS by an average of 3% and by up to 9%. We show that this performance improvement is achievable because frequently misspeculating asserts are identified and disabled by our simple assert monitoring mechanism. Likewise, we find that the compilation overheads introduced by atomic regions are minimal. Finally, we show that atomic regions do not suffer from static code bloat problems and generally reduce static code size.

Figure 6 shows the performance of three configurations of our atomic region implementation in CMS, normalized to the runtime of the baseline configuration. The first configuration shows the performance of a system without our assert monitoring mechanism enabled. The second configuration enables the assert monitoring mechanism but does not use assert operations when speculating on biased exits (*i.e.*, biased-exit branches are simply re-targeted at rollback and recovery code). The third configuration is a complete implementation of atomic regions, including assert monitoring and speculative conversion of biased exits into assert operations.

The complete atomic regions implementation provides an overall performance improvement in nearly every benchmark and none of the benchmarks exhibited a slowdown. Atomic regions provides a 3% average improvement over the CMS baseline that uses superblocks. Seven of the benchmarks exhibit greater than a 2% performance improvement and three of these exhibit a greater than 5% performance improvement. The benchmark with the largest performance improvement is *vortex* at 9.3%.

The performance improvements exhibited roughly correlate with the percentage of dynamic branches which are considered highly biased. Shown in the first column of Table 3 are the percentage of branches executed in each benchmark which are 99.999% biased or greater (*i.e.*, branches for which fewer than 1 in 100,000 dynamic instances oppose the bias). The eight benchmarks with greater than 10% of their executed branches being biased exhibit a performance a performance improvement of 2% or more (with the exception of *eon* which improves by 1.7%). Similarly, the three benchmarks which exhibit a 5% or greater performance improvement have greater than 25% of their branches being biased.

Targeting a 99.999% bias threshold rather than a 100% bias threshold is important to broadening the set of branches which will be considered profitable speculation candidates. Not doing so can have a significant impact on the performance achievable by eliminating profitable opportunities. For example, when only 100% biased branches are considered profitable the percentage of viable dynamic branches in *gap* drops to 11.2% and the performance improvement achieved reduces to 2.3% (from 5.9%).

Using our current implementation of atomic regions, we are unable to profitably speculate on branches less than 99.999% biased, due to a high cost for misspeculation. Shown in the second column of Table 3 is the estimated cost of an assert misspeculation in each benchmark, which we measured using an instrumented version of CMS. In general, recovering from an assert misspeculation takes thousands of cycles. Therefore, it is only worthwhile to speculate on branches that go against their bias significantly less than one out of every thousand executions. Our selection of assert thresholds, shown in Table 2, satisfies this goal although the thresholds have not been highly tuned.

The high cost of an assert misspeculation can also cause severe performance degradation in a naive implementation of atomic regions. The first configuration in Figure 6 shows the performance lost if frequently misspeculating asserts are left untended. If problematic asserts are not disabled, such a configuration will incur a misspeculation once every thousand cycles on average. Combined with the high cost for misspeculations this results in a greater than factor of two slowdown for most benchmarks.

The simple assert monitoring mechanism introduced in Section 4.3 is sufficient to identify problematic asserts so that they can be disabled after retranslation. As shown in Table 3, this simple mechanism is able to reduce the misspeculation rate to fewer than one misspeculation every ten million cycles. In doing so, approximately one in five asserts are disabled through retranslation.

Benchmark	Dynamic Biased Branches (%)	Misspeculation Cost (cycles)	Misspec. / 1M cyc. (no monitoring)	Misspec. / 1M cyc. (w/ monitoring)	Static Asserts Disabled (%)	Static Code Reduction (%)	Static Asserts Eliminated (%)
gzip	4.3	1790	1337	0.06	33.0	-0.1	0.9
vpr	9.8	6188	87	0.08	1.1	0.6	1.2
gcc	5.8	1816	575	0.1	4.5	0.9	1.2
mcf	8.1	1167	968	0.07	24.0	0.4	1.2
crafty	14.9	2985	1529	0.06	28.7	0.4	0.7
parser	14.3	2135	604	0.1	30.1	0.2	2.2
eon	13.0	961	1127	0.04	21.4	0.6	1.1
perlbmk	30.7	1878	942	0.08	17.9	1.9	2.1
gap	26.9	1738	1879	0.08	16.1	1.4	1.0
vortex	38.7	1863	1236	0.1	3.6	3.5	10.1
bzip2	19.0	1179	1497	0.09	34.2	0.4	0.9
twolf	13.6	2598	252	0.07	9.4	0.6	1.0
average	16.6	2195	1003	0.08	18.7	0.9	2.0

Table 3. Atomic region statistics. Lists the percentage of dynamic branches that are 99.999% biased or greater, the estimated misspeculation cost, misspeculation rates (with and without assert monitoring), the percentage of static asserts disabled because they are misbehaving, the static reduction in translation code size enabled by atomic regions, and the percentage of static asserts that have been redundancy eliminated.

However, the additional retranslation costs are minor. The second configuration in Figure 6 measures the performance costs associated with atomic regions by enabling assert monitoring and retranslation but disabling all the optimization benefits of assert operations. The performance costs never exceed 1.5% and generally amount to less than 1% of overhead. Overall, we find that our simple assert monitoring mechanism is sufficient and perhaps conservative.

We also analyze the impact of atomic regions on static code characteristics. Whereas superblocks can incur significant code bloat due to transformations such as tail duplication, atomic regions do not. As Table 3 shows, static translation size generally decreases by a small amount. The reduction in static code is mostly the result of extra classical optimization opportunities exposed by atomic regions. The redundant assert elimination optimizations described in Section 4.4 are also beneficial as they are able to eliminate 2% of asserts on average (up to 10% in *vortex*).

Our experimental results demonstrate that atomic regions are able to offer significant performance improvements on a real machine and that these performance improvements can be achieved without detrimental side-effects. In addition, it has also served as a motivation for future work. As already mentioned, the high misspeculation cost prevents our implementation from considering branches which are less than 99.999% biased. However, if the misspeculation cost could be reduced significantly, it should be possible to target a lower bias threshold and thereby broaden the set of branches that are profitable to convert into asserts.

To reduce the misspeculation cost, we plan to implement a misspeculation recovery mechanism that redirects execution to a non-speculative translation rather than the CMS interpreter. Doing so will incur some code duplication, but so long as the duplication is incurred judiciously it could make for a worthwhile trade-off.

6. Related Work

Several other compiler abstractions have previously been proposed, of which three have achieved widespread adoption. Trace scheduling was the first to introduce the notion of an optimization unit that crosses basic block boundaries and can easily incorporate profile information[6]. In practice, optimizing across side-entrances in a trace introduces bookkeeping complexities that prove difficult to manage. The superblock solved this problem by introducing a single-entry multiple-exit compilation unit, free of reconvergent control flow[9]. The hyperblock furthers the superblock by including small control flow "hammocks" by predicating them[13].

Regardless of the compilation abstraction chosen, speculative compiler optimizations must guarantee correct program execution of speculatively optimized code. Guaranteeing that speculatively executed operations will not adversely affect correctness by computing incorrect values or triggering spurious exceptions requires sophisticated analysis and severely limits optimization opportunities. Therefore, a range of hardware proposals have been proposed to aid the compiler.

Instruction boosting[19] supports speculatively hoisting operations above branches by enabling the compiler to communicate its decisions to the hardware. The result of a speculatively hoisted operation is labeled with the future branch outcomes upon which it is control-dependent. The hardware uses this information—in conjunction with shadowed register files and store buffers—to only commit the result or any raised exception once the operation becomes non-speculative.

Sentinel scheduling[12] and write-back suppression[3] provide hardware support to simplify exception handling for speculatively hoisted instructions. In sentinel scheduling, instructions which may speculatively cause an exception are annotated and are paired with another instruction that non-speculatively checks for exceptions. With write-back suppression, all speculatively executed instructions are annotated so that if one raises an exception all speculative updates can be suppressed.

All three of these schemes specifically focus on handling speculative results and speculative exceptions and simply ignore other operations. As a result misspeculation recovery remains a complex and non-trivial problem because recovery code must be generated for a multitude of possible executions. Both CMS and atomic regions differ because they utilize hardware atomicity rather than explicitly annotated speculative instructions. Hardware atomicity enables simple recovery because if a misspeculation occurs hardware rolls back *all* state to a well-defined point. Execution is then simply redirected to a non-speculative implementation of the same code region.

The rePLay framework[15] converts biased control-flow into hardware assert operations which enable the removal of side-exits from an optimization region. A hardware-only system, rePLay relies on a modified branch predictor to identify predictable instruction traces and place them into atomically executed blocks, called frames, which are processed by a hardware code optimizer. These optimized frames are then stored in a frame cache, and execution is redirected to them by a frame predictor. If an assert detects a mis-

speculation, the frame is rolled back, and execution resumes using normal instructions.

In comparison, atomic regions require only a fraction of the hardware needed by rePLay. The tasks of identifying speculative opportunities and generating optimized code for them is all handled by a software optimization system. Likewise, the handling of optimized code is left to software and the code itself is stored in conventional physical memory. Furthermore, an atomic region, which can contain arbitrary control flow, is more general than a frame, which can only have a single-exit. This enables atomic regions to be used in larger optimization scopes and, thereby, expose more speculative opportunities.

The hardware required for atomic regions is nearly identical to hardware proposed in other contexts. Atomic execution hardware forms the substrate for techniques such as speculative lock elision[16], removing penalties of strongly-ordered memory models[2], and transactional memory[11] and shares many similarities to memory system support for speculative multithreading[8].

7. Conclusions

In this paper, we have demonstrated that our previously proposed atomic region abstraction truly is simple and intuitive to integrate into a mature compilation system. We have also shown that atomic regions expose real performance opportunities even in a well-engineered commercial system.

Our experience in this work has also led us to form several opinions on the relative merit and utility of atomic regions, especially in comparison to superblocks. Our previous view of atomic regions and superblocks as purely competitive abstractions was overly simplistic. Instead, in our experience atomic regions and superblocks are complementary and synergistic with one another.

Specifically, the key advantage of the atomic region abstraction lies in exposing opportunities to remove operations that are partially redundant or partially dead along hot paths (*i.e.*, operations that are fully redundant or fully dead once cold paths are removed). The strength of the approach is the relative simplicity in which it can expose these opportunities. Although superblocks could be used to expose such optimization opportunities, doing so requires a significantly larger effort. Despite being a mature and highly-engineered implementation, CMS does not exploit superblocks for partial redundancy or partial dead code elimination.

These observations have led us to believe that the real benefits of the superblock abstraction are scheduling optimizations that reduce critical path height and increase instruction-level parallelism. On a wide in-order superscalar such as Efficeon, generating good static schedules is key to performance and therefore the superblock plays a critical role on these types of machines.

Looking forward, physical constraints portend a re-emergence of simple, in-order processor designs. To provide good single-thread performance, these designs necessitate sophisticated compiler infrastructures, and we believe that hardware support for speculative optimizations is an energy and complexity effective approach to achieving that performance. Specifically, we believe support for the atomic region, along with the superblock, is a strong candidate for incorporation into these future designs.

8. Acknowledgments

This work was performed while the first author was on internship at the Intel corporation. We would like to thank Pierre Salverda, John Kelm and Jeff Cook for their feedback on this paper. We would particularly like to thank the numerous developers of the Efficeon processor and of CMS whose forward thinking and creative designs made our own work possible

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th International Symposium on Computer Architecture*, pages 233–244, 2009.

[3] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W.-m. W. Hwu. Speculative execution exception recovery using write-back suppression. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 214–223, 1993.

[4] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[5] J. C. Dehnert et al. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, 2003.

[6] J. A. Fisher. Trace scheduling: a technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[7] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel®Pentium®M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2):21–36, 2003.

[8] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 195, 1998.

[9] W. M. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1):229–248, Mar 1993.

[10] A. Klaiber. The Technology Behind Crusoe Processors. Transmeta Whitepaper, Jan. 2000.

[11] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, Dec. 2006.

[12] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Trans. Comput. Syst.*, 11(4):376–408, 1993.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *In Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[14] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 174–185, 2007.

[15] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[16] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, 2001.

[17] G. Rozas. Memory management methods and systems that support cache consistency. United States Patent 7,376,798, May 2008.

[18] G. Rozas, A. Klaiber, D. Dunn, P. Serris, and L. Shah. Supporting speculative modification in a data cache. United States Patent 7,225,299, May 2007.

[19] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, 1992.

[20] C. Zilles and N. Neelakantam. Reactive Techniques for Controlling Software Speculation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 305–316, 2005.