BlockChop: Dynamic Squash Elimination for Hybrid Processor Architecture

Jason Mars
University of Virginia
jom5x@cs.virginia.edu

Naveen Kumar
Intel Corporation
naveen.j.kumar@intel.com

Abstract

Hybrid processors are HW/SW co-designed processors that leverage blocked-execution*, the execution of regions of instructions as atomic blocks, to facilitate aggressive speculative optimization. As we move to a multicore hybrid design, fine grained conflicts for shared data can violate the atomicity requirement of these blocks and lead to expensive squashes and rollbacks. However, as these atomic regions differ from those used in checkpointing and transactional memory systems, the extent of this potentially prohibitive problem remains unclear, and mechanisms to mitigate these squashes dynamically may be critical to enable a highly performant multicore hybrid design.*

In this work, we investigate how multithreaded applications, both benchmark and commercial workloads, are affected by squashes, and present dynamic mechanisms for mitigating these squashes in hybrid processors. While the current wisdom is that there is not a significant number of squashes for smaller atomic regions, we observe this is not the case for many multithreaded workloads. With region sizes of just 200 – 500 instructions, we observe a performance degradation ranging from 10% to more than 50% for workloads with a mixture of shared reads and writes. By harnessing the unique flexibility provided by the software subsystem of hybrid processor design, we present ***BlockChop****, a framework for dynamically mitigating squashes on multicore hybrid processors. We present a range of squash handling mechanisms leveraging retrials, interpretation, and retranslation, and find that BlockChop is quite effective. Over the current response to exceptions and squashes in a hybrid design, we are able to improve the performance of benchmark and commercial workloads by 1.4x and 1.2x on average for large and small region sizes respectively.*

1 Introduction

The evergrowing need for energy efficient, yet high performing, processor microarchitecture designs continues to compel computer architects to develop innovative technologies that excel in both objectives. One such technology that is receiving a notable amount of renewed attention from industry and academia is the *hybrid processor* design. As shown in Figure 1(a), a *hybrid* microarchitecture design is one that uses hardware/software co-design to couple a complex software *binary translation* (BT) subsystem with a simple, often in-order, underlying hardware design. The underlying hardware implements a custom native ISA designed specifically to enable and support the BT subsystem. The BT subsystem dynamically translates the host ISA to the native ISA and is responsible for much of the complexity in the processor such as dynamic instruction scheduling, load/store reordering, speculation, and aggressive optimization. The most notable commercial examples of such a design are the Crusoe [13] and Efficeon [23] processors produced by Transmetta Corp, although there has been other work involving hybrid processor designs such as Daisy [15], PARROT [27], and rePlay [25]. As of today, only uniprocessor hybrid designs have been realized. However, for such a design to be commercially viable, hybrid designs must keep up with the growing core counts of today's production processors.

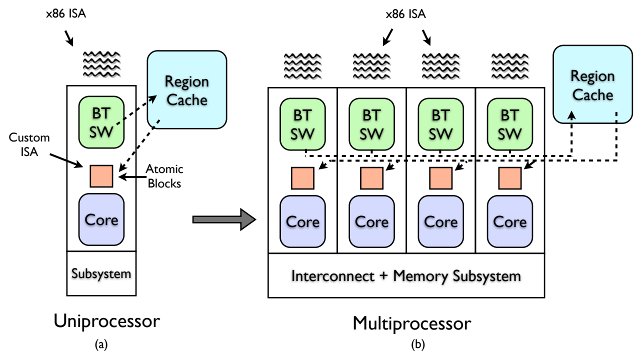

Figure 1: Transitioning the Hybrid Architecture from Uniprocessor to Multiprocessor

Hybrid processors leverage hardware atomicity as an integral primitive for optimization. The instruction scheduling, speculative load/store reordering, and aggressive optimizations performed by the BT component of the microarchitecture occurs within the bounds of regions called *translations*. Translations are static regions of application code. By leveraging *conditional commits* large atomic regions can be formed dynamically. We refer to the sequence of dynamic instructions executed between two commit points within a translation as *atomic blocks*. On a uniprocessor design, these atomic blocks abort and rollback when a speculative condition is violated, resulting in the costly consequence of the BT subsystem resorting to an interpretation mode for the corresponding host instructions. However, the BT subsystem is tasked to form only those translations that are likely to successfully commit, and in practice, *squashes* and *rollbacks* due to mis-speculation occur infrequently for a uniprocessor design. Atomic regions is one of the of the key strengths and enabling technologies of the hybrid design in that it enables aggressive speculative optimizations. However, when moving to a multicore design, this strength may become a weakness.

As we move to a multiprocessor, shown in Figure 1(b), a new condition for squashes and rollbacks emerges. When multiple atomic blocks are simultaneously executing, data elements shared across cores may be read and written to during the speculative execution of the regions. These data conflicts, or *squash hazards* [1], violate the atomicity requirement of the atomic block and at least one block must be aborted on a conflict. Understanding the magnitude of this potential problem for hybrid processors, and how the flexibility of the hybrid design can be utilized to mitigate the performance implications of these squash hazards is critical for steering the design of a multicore hybrid processor.

Recent work has looked at this problem for blocked-execution architectures and concluded that for small region sizes (less than 2K instructions) squashes are infrequent [1, 8, 19]. While this may be the case for benchmark workloads that have a high number of shared reads and few shared writes (such as Parsec and Splash), it remains unclear whether this is the case for other workloads and commercial applications. In addition, there is no prior work investigating how atomic blocks, formed by the processor itself, can be dynamically reformed to minimize and/or eliminate squashes. In this work, we address both of these questions.

In this paper, we investigate the problem of data conflict squashes on a dual and quad-core hybrid design across a range of benchmark and commercial workloads. We also introduce **BlockChop**, a framework for the dynamic mitigation of data conflict squashes in hybrid architecture designs. BlockChop leverages the unique capabilities of a hybrid processor design to observe and respond to squashes as they occur dynamically. These responses span a range of squash handling mechanisms that use techniques such as retrials, interpretation, and re-translation to automatically and dynamically identify and eliminate squashes.

The specific contributions are as follows:

- We perform an investigation of the severity of shared data conflicts for a range of small and large regions sizes across benchmark and commercial workloads. We find that even at smaller region sizes, shared data conflicts are indeed quite problematic for hybrid processor designs.

- We describe **BlockChop**, the first squash handling framework for hybrid processors. BlockChop provides a platform for designing adaptive policies to automatically and dynamically mitigate and eliminate squash hazards.

- We identify five squash handling mechanisms (SHMs) that leverage retrials, interpretation, and re-translation to respond to squashes dynamically. We also provide a comparative analysis of these SHMs and discuss the particular scenarios where each proves most appropriate.

Through our investigation of the severity of squashes due to shared data conflicts on our hybrid design, we observe a significant performance penalty. With region sizes of just 200 – 500 instructions, we observe a performance degradation ranging from 10% to more than 50% for workloads with a mixture of shared reads and writes. When using BlockChop to dynamically and adaptively mitigate these squashes we observe significant performance gains. Over the state-of-the-art response to exceptions and squashes in a hybrid design, we are able to improve the performance of benchmark and commercial workloads by 1.4x and 1.2x on average for large and small region sizes, respectively.

The remainder of the paper is organized as follows. In Section 2, we provide background on hybrid processor design. We then discuss how shared data conflicts impact a multicore hybrid design in Section 3. We introduce BlockChop and several squash handling mechanisms in Section 4. In Section 5, we investigate the severity of squashes for benchmark and commercial workloads. Section 6 presents related work, and finally we conclude in Section 7.

2 Hybrid Processor Design

The most notable examples of hybrid processor designs are the Transmeta's Cruseo and Efficeon processors [13, 23]. Throughout this work, we use the Efficeon processor and its binary translation subsystem, the Code Morphing Software (CMS), as a basis for a multicore hybrid processor. In this section, we describe the design of the hardware and software components of Efficeon, and discuss a multicore design.

2.1 Processor Architecture

The primary design objective of the Efficeon is to use a simple in-order low-power custom processor coupled with a sophisticated binary translation software subsystem to enable aggressive out of order rescheduling and optimization. While the simplicity of Efficeon hardware provides low-power advantages in this design, the adaptive binary translator, CMS, is responsible for delivering high performance.

2.1.1 Core Architecture

The Efficeon is a *very long instruction word* (VLIW) processor. Each VLIW instruction, or *molecule*, consists of up to eight 32-bit *packets*. A packet encodes functional operations, called *atoms*, but can also encode some auxiliary information, such as a memory alias protection marker that allows CMS to signal to the hardware when it speculatively reorders and elides memory operations. Efficeon's atoms use a traditional three-address format and can be assigned to one of seven functional units: two ALU, two memory, two floating point and a branch unit. Efficeon hardware has few hardware interlocks. In fact, the only interlocks are to handle variable latency memory operations. CMS guarantees correctness by carefully scheduling operations, inserting nops as necessary. While the lack of interlocks significantly simplifies hardware, it does imply that CMS must conservatively schedule instructions across control boundaries. Efficeon's custom ISA uses 64 general purpose integer registers and 32 floating point registers. The register file is much larger than the number of architectural registers in the x86. This large register file permits dedicated native registers for maintaining x86 architectural state, while leaving plenty of registers for CMS to use. The core architecture of Efficeon also provides memory alias detection hardware to guarantee the correctness of memory operations that has been reordered by CMS.

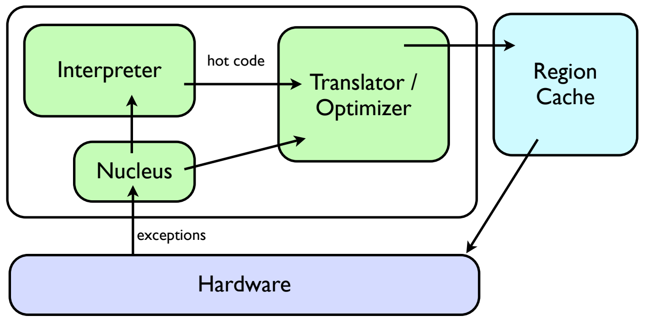

Figure 2: Software BT Subsystem

2.1.2 Hardware Atomicity and Speculation Support

Efficeon provides hardware support for speculation, including support for fast checkpointing and rollback of registers and memory. Checkpoint/rollback of registers is made possible by having two sets of register files, a working copy and a shadow copy. Memory checkpointing/rollback is similarly realized by associating a speculative bit in the first level data cache. Speculative state is written into working copy of registers and by setting the speculative bit in the data cache and marking the lines dirty. If a cacheline is dirty and written to speculatively, its dirty contents are first evicted to a victim cache. An explicit *commit* operation commits all speculative state in a single cycle by copying working register set into the shadow register set and by clearing all speculative bits in cachelines. Similarly, CMS can use a *rollback* operation to roll back speculative state by copying values from shadow register set to working set and invalidating all speculative lines in the data cache. The CMS builds *atomic* regions using these constructs and performs aggressive code optimizations within the atomic regions. If a fault or exception occurs within an atomic region, a fault handler in CMS rolls back speculative state. CMS can also detect atomic regions that fault frequently, often due to aggressive speculation, and then adaptively regenerate non-faulting code. Indeed, CMS makes heavy use of atomicity to aggressively optimize code. Atomicity is the single most important feature in Efficeon for realizing high performance.

2.2 Binary Translation Subsystem

The binary translation and optimization software, such as Efficeon's CMS, is the heart of a hybrid processor. As shown in Figure 2, CMS broadly consists of three components: the **interpreter**, the **translator**, and the **nucleus**. The interpreter and translator are responsible for emulating x86 instructions. Initially, each x86 instruction is interpreted. Frequently executed x86 instructions are then translated and optimized based upon the execution profile collected during interpretation. The nucleus is responsible for handling interrupts and exceptions at the host level. Those that occur at x86 level are handled by the interpreter. The nucleus is also responsible for recovering from speculation faults (i.e., mis-speculation).

The CMS translator is responsible for optimization and scheduling of atoms emulating x86 code. The code regions identified as frequently executed by the interpreter are subdivided into atomic regions and optimized. While the translator is essentially free to arbitrarily classify instruction regions as a single atomic block, typical atomicity boundaries are loop iterations and I/O boundaries, with an upper bound on the number of static basic blocks and instructions.

Instructions within atomic regions can be reordered by the translator without regard for memory consistency and precise exceptions. Efficeon's speculation support and alias hardware is used to avoid x86-visible effects of mis-speculation and imprecise exceptions. Atomic regions that fault due to mis-speculation, or true x86 exceptions, invoke the nucleus. Execution of the atomic region is then rolled back. The CMS interpreter subsequently interprets the respective x86 instructions while carefully maintaining precise exceptions and memory ordering. The nucleus marks and monitors translations that repeatedly fault and adaptively retranslates to avoid further faults.

2.3 A Multicore Design

Our hybrid multicore design targets the space of common x86 commercial multi-processor architectures. The x86 multiprocessor implementation essentially permits multiple independent processor cores to share memory and devices. The instruction set architecture of our multicore design is much the same as the uniprocessor's instruction set, except for a small number of additions to support atomicity and serialization. Communication between processors is accomplished using either shared memory or inter-processor interrupts.

The primary responsibility of a hybrid multi-processor is to efficiently emulate guest (e.g., x86) instructions, including atomic and serializing instructions, under the memory consistency model and other constraints defined by x86. The multi-processor CMS can, however, use a memory consistency model that is stricter than that implemented by traditional x86 multi-processors.

Complications to multi-processor design arise not only from intricacies in emulation of multiprocessor guest ISA, but also from interaction with another copy of CMS executing on another processor core. Some of these inter-CMS interactions are explicit, such as locks and inter-processor interrupts, while others are implicit, such as modification of x86 code on one core rendering translated code used by another core stale. Solutions to the problems of multiprocessor consistency, multi-processor ISA emulation and multi-processor CMS design are beyond the scope of this paper. This paper specifically targets a fundamental problem in the hybrid multi-processor design, namely data cache conflicts.

3 Data Conflicts in a Hybrid Design

As the size of atomic blocks increases, so does the chance for these blocks to suffer shared data conflicts. In this section, we first discuss how large atomic blocks are formed in a hybrid processor, shared data conflicts and the high level semantics that cause them, and how squashes and rollbacks impact a hybrid processor design.

3.1 Larger Blocks in a Hybrid Design

For a hybrid design, larger atomic blocks are desirable for a number of reasons. In a hybrid design, the architectural state of the processor must be enforced at the host ISA level,

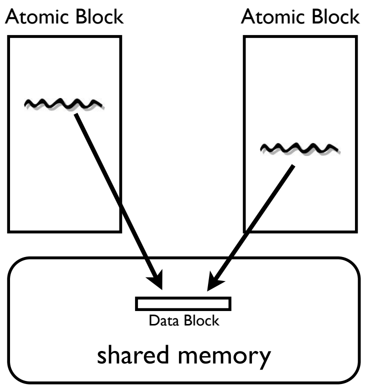

Figure 3: Conflicts for Shared Data Blocks

therefore all speculative optimizations are restricted to enforce the architectural state at every commit point. A larger window of execution between commit points allow for these optimizations to be applied more aggressively. In addition, frequent commits are costly. At every commit point a check for shared conflict is performed by accessing the coherence directory of the last level cache. The more frequently commits occur, the higher the overall cost of performing these checks.

As the unit of optimization in a hybrid design is a translation that typically spans a relatively small number of static instructions, a mechanism must be in place to allow for longer atomic blocks between commits. To achieve this we leverage *conditional commits* [4] where static commits inside the translations are replaced with *branch to skip* instructions that skip the commit until a condition is met. These conditions can be in the form of specialized checks for the availability of speculative resources, or atomic block size. In addition to the insertion of these branch to skip transformations, additional transformations are applied to allow the optimization of larger regions. These transformations are described in prior work [4].

Leveraging *conditional commits*, hybrid processors can form atomic blocks of 10s to 1000s of instructions. However, with larger regions comes an increased chance to suffer squashes due to shared memory data conflicts.

3.2 Shared Data Conflicts

Figure 3 illustrates a shared data block conflict. In this figure two concurrently executing atomic blocks are accessing the same data block in shared memory. When both of these accesses are reads, execution can proceed and the subsequent commits are allowed. However, if a write operation is performed by either core, this data block moves from shared state to exclusive state for the first core performing the write. If other concurrently executing blocks either read or write to this memory block, the atomicity assumption is violated. To ensure faithful enforcement of the memory consistency model of the guest ISA in the example shown in Figure 3, one of these regions must be squashed.

With larger atomic blocks, both the chance of suffering a squash, and the performance cost of enduring the squash, increases. Upon a squash, a software exception handler is invoked to perform a rollback, and all of the forward progress made from the prior commit point must be rolled back. With larger regions, more time is available for other concurrent atomic blocks to perform a write to any one of the data blocks accessed, and a larger amount of work is wasted on a squash.

3.3 Source of Shared Data Conflicts

There are a number of high level constructs that are responsible for shared data conflicts in multithreaded applications.

1. **[Benign and Buggy Data Races]** When two or more threads modify the same memory element that is not protected by synchronization, we have a data race. Although it is not good programming practice to have data races, these may be benign in that they do not affect the correct execution of an application. Programs with these data races in the critical path are prone to suffer shared data conflicts as the ordering of memory operations from the hybrid processor must be conformant with the memory consistency model of the guest ISA.

2. **[Lock Variables]** Potentially the most common cause of shared data conflicts are the usage of lock variables. When locks are released and acquired, this involves shared reads and write to the data block where the lock variable resides.

3. **[Lock Free Data Structures]** Lock free data structures are used heavily in a number of applications. These data structures provide thread-safe access to shared data without the use of explicit synchronization operations. Fine grained reads and writes to these data structures can safely occur from a multiple threads and thus shared data conflicts can occur.

4. **[Dynamic Linkage]** Dynamic linkage constructs such as the Procedure Linkage Table (in Unix) provides a level of indirection to facilitate dynamic linking. For the standard C library, there are over 600 entries in this table. This shared table is read and modified as library functions are called. There are a number of application and implementation level properties that impact the degree of access and usage of the PLT throughout execution. If it is used heavily and updated frequently, shared data conflicts are likely to be problematic.

5. **[Library Code]** Library function calls such as `malloc`, or `signal` themselves use lock variables and update global data structures. Frequent use of these types of library functions can lead to shared data conflicts.

6. **[False Sharing]** The current architectural view of the memory system is as a data block / cache line granualirty. The distinction between individual words in a single line is not made by the coherence and memory subsystem protocols. Therefore, if multiple threads are accessing different words on the same line, the whole line is marked dirty, leading to shared data conflicts.

3.4 Squashes and Rollbacks

In our Efficeon-based hybrid design, data conflicts are detected lazily on a commit. If a conflict is detected, the committing core will send a squash signal to all cores for which it

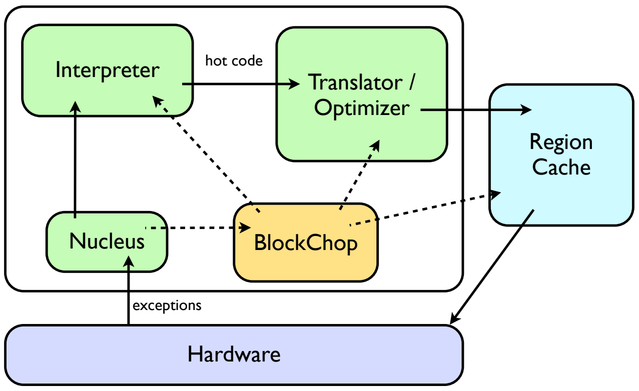

Figure 4: The BlockChop Framework

is in conflict. When a squash occurs, the squashed core transitions to an interpreter mode where the host ISA instructions are interpreted one by one. As squashes on a uniprocessor occur only due to exceptions and mis-speculated optimizations, these squashes are infrequent and results in a negligible performance impact in an application's steady state. If squashes due to shared conflicts are also infrequent in practice, responding to these squashes with the interpreter may be a reasonable response. Considering that atomic blocks in hybrid processors does not approach the large sizes of up to 25,000 instructions as studied in prior work [1], this simple solution may be sufficient. As we show in Section 5, we find that for the class of applications that have a mixture of shared reads and writes, this is not the case, and better *squash handling mechanisms* are needed to mitigate and eliminate squash hazards.

4 Squash Elimination with BlockChop

The atomic blocks formed by a hybrid processor is unlike those in the transactional memory (TM) programming paradigm. In a TM system, atomic regions are formed by the developer, and the scope of these regions can not be fundamentally changed by the underlying processor. In a hybrid processor, these regions are formed by the processor at a lower level of abstraction. As such, these regions can be reformed and adapted by the processor itself, and a wider range of responses to high conflict regions is available to the software component of the hybrid processor.

4.1 The BlockChop Framework

To address the problem of frequent squashes in a hybrid design, we present the *BlockChop* squash handling framework. As shown in Figure 4, BlockChop is an extension to the BT subsystem of hybrid processor to enable adaptive, software controlled, squash management. One of the key advantages of the hybrid design, is the ability of the BT subsystem to observe application behavior and employ feedback directed and situation specific execution and optimization policies. BlockChop leverages this capability to direct its dynamic response to squashes.

As shown in Figure 4, upon a squash exception, the nucleus is triggered. If the squash is due to a data conflict, the nucleus then recovers, rolls back execution, then passes control to the BlockChop framework. BlockChop then decides how to handle the squash and may then decide one of three options.

1. **[Interpreter]** Pass control to the interpreter to guarantee squash-free interpreted execution of the atomic block.

2. **[Translation]** Pass control to the translator to apply a translation to reduce the likeliness of suffering another squash. In this work, we have designed and implemented a *"chopper"* translation.

3. **[Region Cache]** Directly re-execute the same region in hopes that it does not conflict again.

BlockChop can also maintain historical information on which to base its squash response. This information includes, but is not limited to, translation squash frequencies, squash profiles of individual instructions, and other statistical information.

4.2 Squash Handling Mechanisms

BlockChop provides a platform for the creation of dynamic and adaptive squash handling mechanisms. In this section, we describe five response heuristics using BlockChop to respond to squashes as they occur throughout execution, summarized in Figure 5.

4.2.1 Interpret

The first heuristic shown in Figure 5(a) show the interpreter based response. Upon a squash, BlockChop simply passes control to the interpreter for non-squashable execution of the atomic block. This approach is the current default for squashes in the Efficeon hybrid processor for exceptions that interrupt the execution of an atomic block. We use this approach as the baseline throughout our evaluation. While squashes on the interpreting core is guaranteed not to occur, interpretation is slow. On the Transmeta Efficeon with a pre-release version of CMS 7.0 (never released to market), each x86 host instruction is interpreted at the cost of 10s to 100s of cycles on average.

4.2.2 Delayed Retry

Another intuitive response is to simply retry the same atomic block, as shown in Figure 5(b). Assuming a retry at $delay = 0$ is successful and the atomic region does not squash again, this response is the most efficient. However, if we suffer retrials throughout execution, we have a sustained cost that is proportional to the size of the atomic blocks. A $delay$ can be set to try to offset the timing of conflicting accesses to increase the likelihood of avoiding the squash, however at large atomic block sizes this would prove less effective when conflict prone instructions are in the critical path of the application. In addition, if retrials are unsuccessful when large delays are used, the cost of the squashes are further exacerbated.

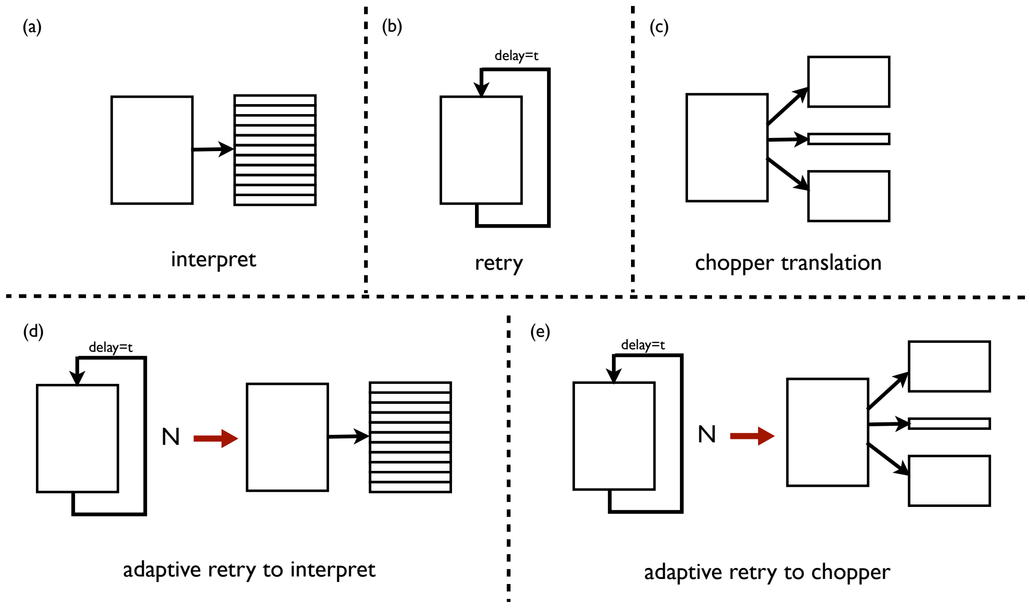

Figure 5: Squash Handling Mechanisms

```
chopper (excep_pc):

// roll back architectural state
rollback()
// retrieve descriptor for the translation
tr_desc <— lookup_translation(excep_pc)
// identify conflicting "guest" instruction
x86_eip <— map_to_x86(tr_desc, excep_pc)
// mark conflicting eip for isolation
tr_desc.retrans_isolate_eip <— x86_eip
// retranslation isolates eip in its own commit region
retranslate_and_dispatch(tr_desc)
```

Figure 6: Pseudo Code for Chopper Translation

4.2.3 Chopper Retranslation

The chopper translation, illustrated in Figure 5(c), is a re-translation approach designed to eliminate squashes by isolating instructions that are responsible for squashes. This translation algorithm, shown in Figure 6, inserts `commit` instructions before and after memory access operations that cause squashes, effectively chopping the atomic block. Note that our coherence protocol has been enhanced to contain the program counter for the instruction initiating the cache access. This program counter is delivered to the processor where squash occurs in a machine specific register. For this approach to be most effective, instead of inserting *conditional commits*, it marks the instruction pc as a *serializing* instruction and the call to `retranslate_and_dispatch` inserts a regular commit before and after the instruction. By inserting non-conditional commits inside the critical path within a translation, the dynamic atomic block is not chopped into two pieces, it is effectively chopped at every occurrence of the chopped instruction.

It is important to note that re-translation includes complex compilation analyses in a hybrid design, and as such, it is costly, taking 10s-100s of thousands of cycles depending on the number of static instructions spanning the atomic block. The major advantage of this re-translation approach is that it neutralizes squashes for the duration of execution, and the initial cost of retranslation is quickly amortized during execution.

4.2.4 Adaptive Interpret

Another policy may be to use historical information to combine retrials with interpretation. This approach is shown in Figure 5(d). As execution occurs, BlockChop first takes a retry approach to handling squashes. The success or failure of retrials are observed by BlockChop and if BlockChop observes a number of consecutive failures, a response of interpretation is taken. The rationale behind this approach is that retrying is likely to fail after a number of failed attempts has been made. Passing control to the interpreter at this point guarantees forward progress. We also have the added benefit of eliminating the possibility of starvation.

4.2.5 Adaptive Chopper

Similar to adaptive interpret, adaptive chopper, shown in Figure 5(e), resorts to the chopper translation after a number of failed retries. The core algorithm for adaptive chopper is shown in Figure 7. Instead of immediately chopping any conflicting instructions as they occur, we neutralize problematic memory operations only as they demonstrate their persistence by continually causing squashes. The tradeoff between adaptive interpret and adaptive chopper is whether to reinterpret and guarantee forward progress now at a cost, or neutralize the propensity of particular instructions to cause squashes for the long term at an even larger initial cost. Another advantage of this adaptive approach is the retrials themselves serve to filter out only those operations that are most conflict prone. As we show in Section 5, this adaptation significantly reduces the number of chops applied to the application.

```
adaptive_chopper (excep_pc, d_addr):

//roll back architectural state
rollback();
//retrieve descriptor for the translation
tr_desc <— lookup_translation(excep_pc)

//is this a retry
if !frwd_progress() && tr_desc.d_conflict.addr == d_addr:
  //is the retry count greater than "threshold"
  if tr_desc.d_conflict.count > threshold
     // identify conflicting "guest" instruction
     x86_eip <— map_to_x86(tr_desc, excep_pc)
     // mark conflicting eip for isolation
     tr_desc.retrans_isolate_eip <— x86_eip
     // retranslation isolates conflicting instruction
     retranslate_and_dispatch(tr_desc)
  else
     tr_desc.d_conflict.count++
     // retry the same atomic block
     dispatch(tr_desc)
else
  tr_desc.d_conflict.addr  <— d_addr
  tr_desc.d_conflict.count <— 1
  // retry the same atomic block
  dispatch(tr_desc)


forward_progress():

// check whether instructions commited
if chopper_insn_count == current_x86_insn_count
  return false
else
  chopper_insn_count <— current_x86_insn_count
  return true
```

Figure 7: Pseudo Code for Adaptive Chopper

Processor	Transmeta Efficeon 2 (TM8800)
Frequency	1.2 GHz
BT Subsystem	CMS 7.0 (pre-release)
Registers	64 integer, 64 FPU
Trans. Cache	32MB
L1 I-Cache	128 KB, 4-way, 64B line
L1 D-Cache	64 KB, 8-way, 32B line
Victim Cache	1 KB, fully-associative, 32B line
L2 Unified Cache	1024 KB, 4-way, 128B line
Main Memory	1 GB DDR-400
Op. Sys.	Linux 2.6.19

Table 1: Efficeon Hardware Modeled

5 Evaluation

In this section, we investigate the magnitude of shared memory data conflicts for a range of benchmark and real workloads. We then evaluate the effectiveness of using BlockChop to apply various squash handling mechanisms over simply resorting to interpretation to handle squashes. But first, we describe our experimental setup and simulation methodology.

5.1 Experimental Setup and Methodology

For our evaluation we use a recently developed HybridMP simulator developed at Intel Corp, shown in Figure 8. HybridMP is a functional x86 simulator with a timing model for hardware atomicity and memory subsystem. The model used to implement hardware atomicity has been derived from a pre-release version of the Transmeta Efficeon uniprocessor shown in Table 1. Cycle latencies for events such as commit, rollback, recovery, interpretation, and re-translation are faithfully modeled. The memory subsystem architecture (private/shared caches, and access to main memory) used is representative of the "uncore" of a Quad Core Intel Nehalem architecture.

workload	description
SM-Stress	SpiderMonkey Javascript Engine: JS_BeginRequest and JS_EndRequest are called continuously on javascript threading contexts by 4 threads
NSPR-Time	Mozilla NSPR Thread Package: The calendar/time api is exercised continuously by 3 threads
MemcacheD	MemcacheD: The memcached_increment() api is exercised continuously by 3 threads.
Spidermonkey	SpiderMonkey Javascript Engine: JS_NewContext, JS_BeginRequest, and JS_DestroyContext is exercised continuously by 3 threads

Table 2: Commercial Workloads

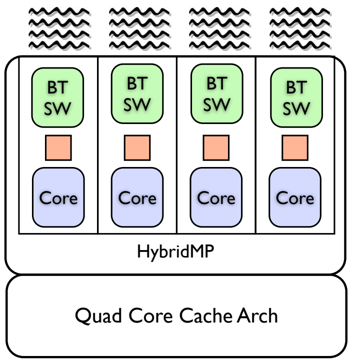

Figure 8: HybridMP Simulation

Within HybridMP, commit points are selected dynamically at a parameterized atomic block size. For example, if HybridMP is parameterized to form atomic blocks of 1000 instructions, a commit is issued every 1000 instructions. This is slightly different than using conditional commits [4] inserted by the BT system in that those commits may be earlier or later than the dynamic commits. However, as each commit point also begins the next atomic block, in the steady state, this simply slightly shifts the location of the commits by few dynamic instructions in practice. It is also important to note that the performance benefit of the added optimization opportunity of larger regions is not included in our evaluation methodology. However, the cost of squashes is significantly more expensive relative to the added optimization benefits of increased region size as entire regions of execution must be squashed and re-executed.

In this work, we look at workloads that have a high amount of shared reads and writes in addition to read only sharing. The workloads used throughout this evaluation span a number of applications from the Stamp [6, 7] benchmark suite using simulator inputs, and the set of commercial workloads described in Table 2. All applications were compiled with GCC 4.6.1 using the `pthread` library. All applications are executed in their entirety. In the cases that are relevant, single thread initialization phases are skipped and only the parallel regions of executions are simulated.

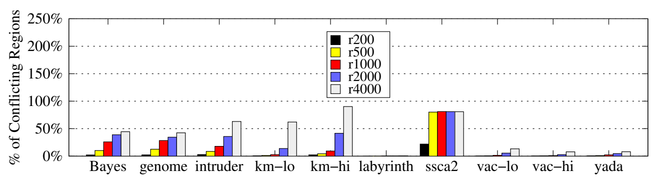

Figure 9: Conflicting Regions (2 Cores)

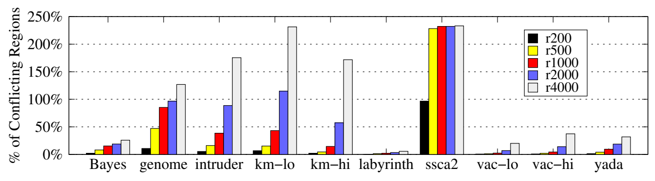

Figure 10: Conflicting Regions (4 Cores)

5.2 Severity of Squashes

First, we investigate the severity across applications with a high degree of shared reads and writes. This section is complementary to prior findings [1], and shows that there are indeed applications that severely suffer from shared data conflicts. In this section, we perform a characterization experiment that only checks for conflicts for every commit region. When a region conflicts with two or more cores, this conflict is counted once per conflicting core. After a conflict is counted the speculative state of the committing region is cleared and is therefore not counted by its conflicting neighbors, preventing conflicts from being counted multiple times across cores.

5.2.1 Benchmark Workloads

Figures 9 and 10 show the percentage of dynamic regions executed that experienced a shared data conflict at five atomic block sizes for both a dual-core and quad-core hybrid design. At each size, a commit is executed after the set region of instructions. Each bar shows the percentage of those commits where a shared conflict is detected.

As shown in these figures, some applications are more prone to shared data conflicts than others. Applications such as `labyrinth`, `vacation`, and `yada` are similar to Parsec and Splash in that they perform less shared read/writes and are thus less likely to have data conflicts. Other applications, such as `ssca2` and `gnome` are much more prone to data conflicts. When moving to a quad core design, the number of conflicting regions increases significantly. Note that the scale of the y-axis has changed to 250%. Keep in mind, however, that even in the cases where we have a small number of conflicts, for a hybrid processor design, there may be a significant performance penalty.

To investigate how the performance of these workloads are affected by their propensity for data conflicts we execute them using a baseline configuration of HybridMP that uses that standard squash handling response. Figures 11 and 12 shows the performance penalty of resorting to the interpreter when a squash occurs. The y-axis shows the normalized IPC relative to execution on HybridMP where atomic blocks are

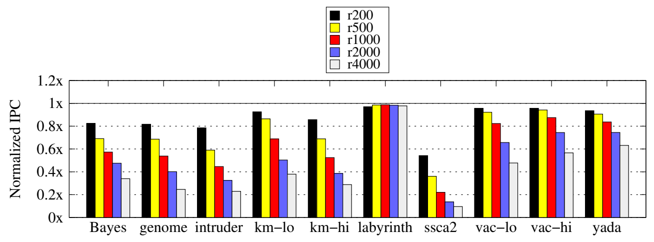

Figure 11: IPC Impact of Squashes on a Dual Core Hybrid Processor (HybridMP)

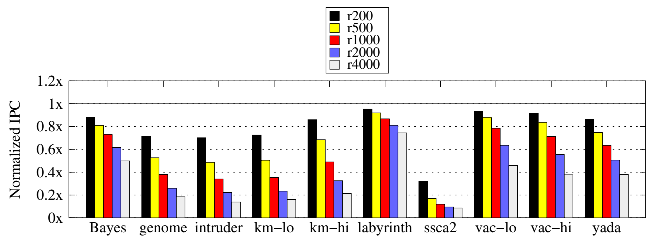

Figure 12: IPC Impact of Squashes on a Quad Core Hybrid Processor (HybridMP)

not used, more specifically, the costs associated with interpretation, commits, and rollbacks are not included.

As shown in these figures, the performance penalty suffered even for applications with a small proportion of conflicting regions is significant. While `labyrinth` is essentially unaffected by squashes, `vacation`, and `yada` suffer more than a 10% degradation as region sizes grows beyond 500. For applications that are prone to data conflicts the performance penalty is overbearing, exceeding 50% in many cases. When moving to a quad core design, these degradations become more severe.

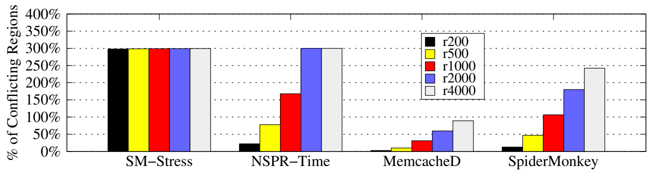

Figure 13: Conflicting Regions (4 Cores)

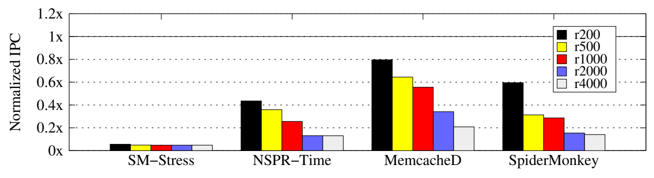

Figure 14: IPC Impact of Squashes on a Quad Core Hybrid Processor (HybridMP)

5.2.2 Commercial Workloads

Benchmark applications are not always representative of real world applications. To address this uncertainty, we

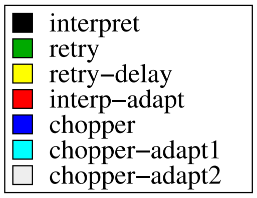

Figure 15: Legend Used for Figures 16 to 35

have also performed our experimentation on four commercial workloads that have shared read and write accesses. These applications are collected from the Radbench benchmark suite [18]. For each workload, a test harness is used to exercise each workload through calls to their relevant APIs as shown in Table 2.

As shown in Figures 13 and 14, we observe the same trend for these commercial workloads. For `SM-Stress` every region results in a squash, even at a region size of 200. This application continually creates and destroys Javascript context. While this type of behavior is not likely to occur throughout execution, this workload represents a phase of execution that exists in real applications. When this type of behavior occurs, a high density of conflicts can be expected.

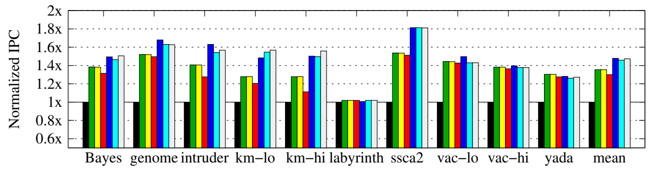

Figure 16: IPC After Applying BlockChop (r4000)

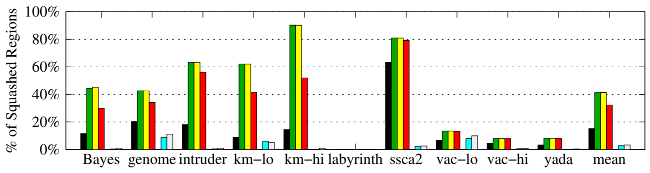

Figure 17: Squashes After Applying BlockChop (r4000)

5.3 Squash Handling Mechanisms

In this section, we perform a comparative evaluation of the various squash handling mechanisms implemented in BlockChop. Seven response heuristics are shown:

squash handler	description
`interpret`	Transition to interpreter.
`retry`	Immediately retry
`retry-delay`	Delayed retry by 30 cycles
`interp-adapt`	Adaptively go to interpreter after 5 failed retrials
`chopper`	Apply the chopper translation on squash
`chopper-adapt1`	Adaptively apply chopper translation after 5 failed retrials
`chopper-adapt2`	Adaptively apply chopper translation after 10 failed retrials

Figures 16 and 17 (see Figure 15 for legend) show the performance improvement (normalized IPC), and percentage of dynamic atomic blocks that were squashed, at a block size

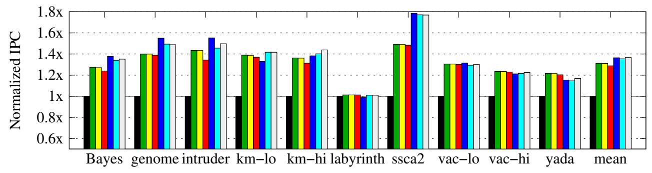

Figure 18: IPC After Applying BlockChop (r2000)

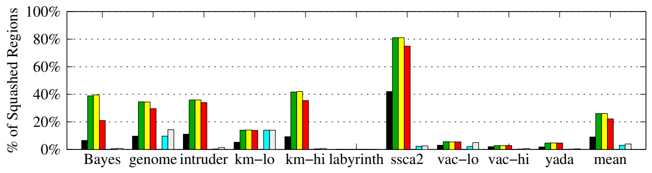

Figure 19: Squash After Applying BlockChop (r2000)

of 4000 instructions when applying each of these various squash handling mechanisms over the baseline of resorting to the interpreter, on a dual core hybrid design. Figures 18 and 19 show this experiment for a block size of 2000.

We observe that, although resorting to the interpretation mode on a squash guarantees forward progress it is consistently the worst of the seven approaches. When in the interpretation mode, each x86 instruction takes 10s of cycles to execute. This cost is high considering interpretation of an atomic block is potentially as costly as 10s of retrials. Even though forward progress is guaranteed, subsequent executions of the atomic block may trip to the interpreter repeatedly. As shown in Figures 17 and 19, reducing or eliminating dynamic squashes does not imply better performance. The method used to reduce the squashes themselves have a cost, for interpretation this cost is significant, so although we have less squashes than retrying, we ultimately have poorer performance. We also observe that for these workloads, although `interp-adapt` is significantly better than the baseline, it is still worse than simply retrying. However, when starvation is likely to occur `interp-adapt` is preferable in that it guarantees forward progress. Our chopper translation proves to provide the greatest gains, both providing higher performance in light of its high costs, and virtually eliminating squashes. Also keep in mind that these workloads are relatively short running as they were run on simulator inputs, in our experimentation we observe that in a long running steady state the chopper heuristic continues to converge to the IPC of having no aborts (with the caveat of enduring more frequent commits).

Figures 20 to 25 show the same experimentation at smaller block sizes ranging from 200 to 1000. Most notably, we observe that at smaller region sizes, retrying is most effective for a number of workloads, and the chopper translation produces much poorer performance, at 200, almost always underperforming the baseline. This effect is due to the high cost of translation vs the higher success rates of retrying when region sizes are small. Here we see the key strength of BlockChop in leveraging the flexibility of a hybrid design, in that, adaptive chopper, `chopper-adapt1` and `chopper-adapt2`, approaches the performance of simply retrying for the smaller regions shown here, while achieving the performance gain of chopping for the larger regions sizes of 2000 to 4000.

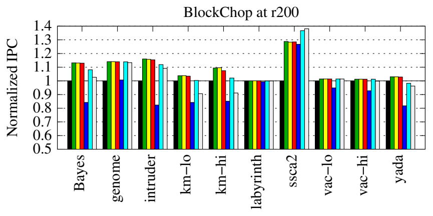

Figure 20: IPC at r200

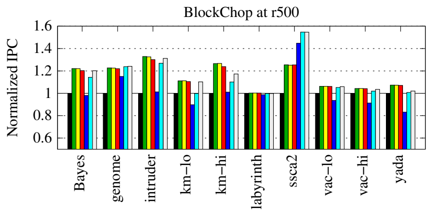

Figure 21: IPC at r500

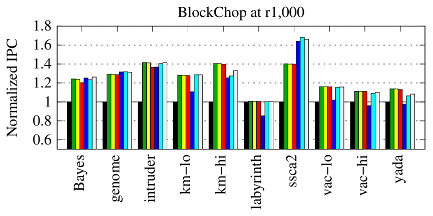

Figure 22: IPC at r1000

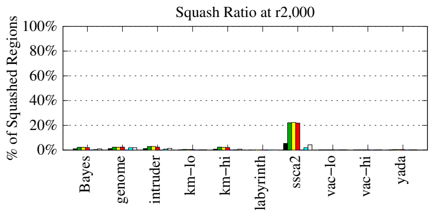

Figure 23: Squashes at r200

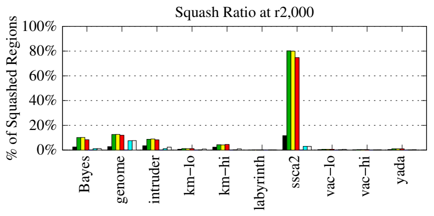

Figure 24: Squashes at r500

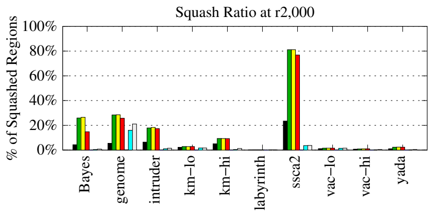

Figure 25: Squashes at r1000

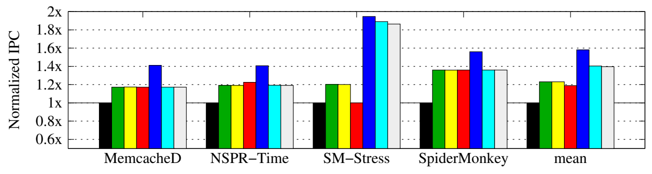

Figure 26: IPC After Applying BlockChop (r4000)

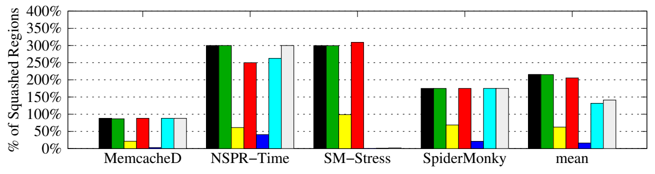

Figure 27: Squashes After Applying BlockChop (r4000)

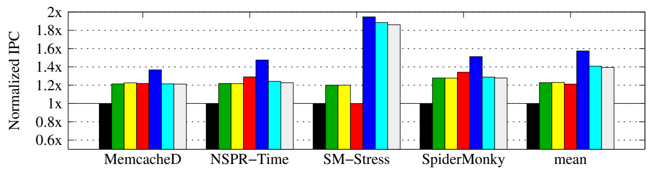

Figure 28: IPC After Applying BlockChop (r2000)

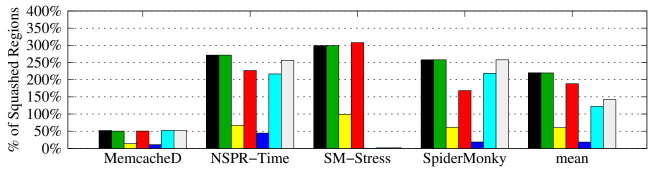

Figure 29: Squashes After Applying BlockChop (r2000)

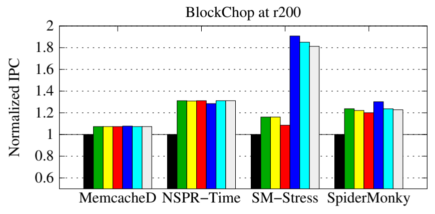

Figure 30: IPC at r200

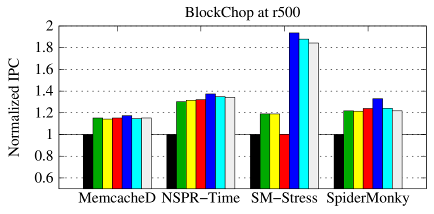

Figure 31: IPC at r500

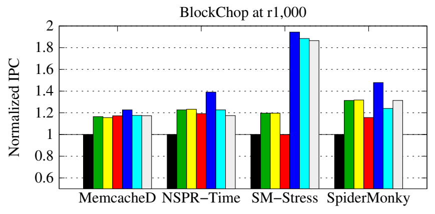

Figure 32: IPC at r1000

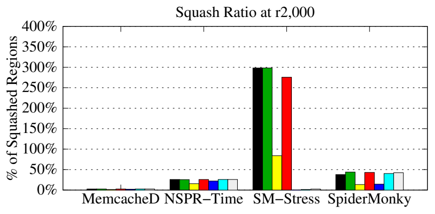

Figure 33: Squashes at r200

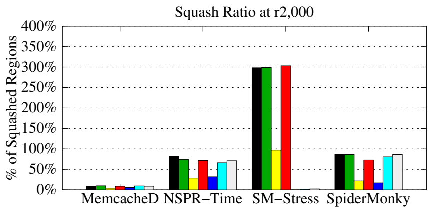

Figure 34: Squashes at r500

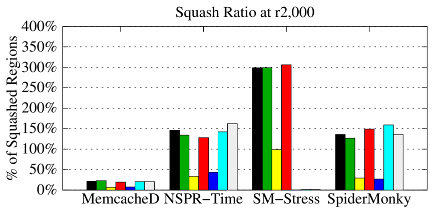

Figure 35: Squashes at r1000

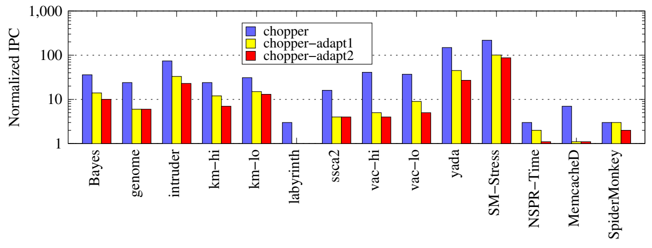

Figure 36: Chop Count

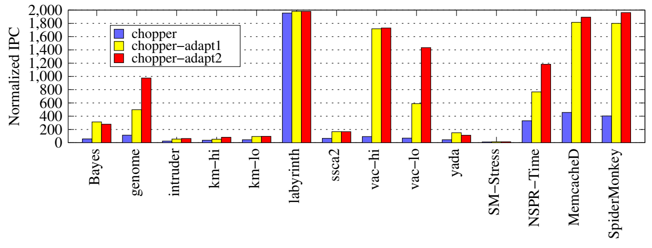

Figure 37: Average Dynamic Region Size

5.4 BlockChop on Commercial Workloads

We have also investigated the effectiveness of applying various squash handling mechanisms on commercial applications on a quad core hybrid design. Figures 26 to 29 (see Figure 15 for legend) show the results for 4000 and 2000 instruction block sizes. Unlike the previous benchmark workloads that are relatively short running on simulator inputs, these workloads have been configured to run for much longer. As a result, we observe a greater benefit of using the chopper translation at larger block sizes as a longer period of execution allows the cost of chops to be amortized over time. As retrials tend to succeed for these runs, the adaptive chopper approaches described tend to trigger retrials in lieu of chopping. For smaller region sizes, as shown in Figures 30 to 35, we observe that chopper does not perform worse than the baseline due to both the fact that the cost of chopping is amortized over long runs, and these applications suffer a higher amount of squashes at smaller region sizes compared to the benchmark workloads presented earlier.

5.5 Characterizing Chopper Translation

To better understand how the chopper translation affects the dynamic execution of the workloads presented we characterize the number of static chops performed during execution and how these chop affects the average dynamic block sizes throughout execution. Figure 36 presents the number of static chops performed for each of the applications across the three chopping heuristics presented, at a block size of 2000. Note that this figure is on a log-scale. As shown in the figure, the retry adaptation effectively functions as a chop filter, allowing BlockChop to only chop blocks are frequently failing during retry. This reduction in the number of static chops both reduces the dynamic cost of chopping, and increases the average atomic block region size throughout execution.

Figure 37 shows the average size of executed dynamic regions through execution across the three chopping heuristics. For some applications such as `intruder`, `yada`, and `SM-Stress`, the average dynamic region size is severely reduced. These small dynamic block sizes occurs as a result of the key conflicting reads and writes being in the critical path of the application's execution. Along the dynamic trace of these workloads, a small number of conflict-prone static instructions appear repeatedly in the trace. These chops have a significant impact on the ability of the BT component of the hybrid system to perform speculative optimizations, however removing squashes is a first order objective as suffering squashes renders any optimization benefit insignificant. For these reasons, the greatest strength of a hybrid processor design, its atomicity support for speculative optimizations, becomes a weakness.

6 Related Work

There has been a sizable amount of work in the area of blocked execution and shared data conflicts. Perhaps the most closely related work is the recent work by [1]. This work explores the very question of squashes due to shared data conflicts and how these conflicts can be mitigated; however, there are a number of key differences. Firstly, this work focuses on much larger block sizes of more than 15,000 instructions per block. At this granularity, squashes become problematic even for mainly read-only applications such as Parsec and Splash. Indeed, real-world applications that demonstrate extensive read-write sharing have not been evaluated. Also, the solutions to mitigate squashes proposed by this work are applied statically and therefore have limited appeal in backward-compatible systems. Our work is complementary to this work in that we show that shared data conflicts can be quite problematic at smaller region sizes for a number of benchmark and commercial applications. We also propose a framework and mechanisms to dynamically and adaptively mitigate and eliminate squashes.

In addition to this prior work, a number of other works [11, 19, 20] has dealt with the issue of shared data conflicts in various types of atomic regions. However, these works do not cover the important question of how atomic blocks impact the design of hybrid multicore architectures.

There has been a significant research effort investigating various issues related to blocked-execution architecture designs [2, 3, 9, 8, 10, 14, 16, 26, 29, 30, 31] that use atomic regions as a core primitive of optimization and execution. There is also a wealth of prior work on hardware / software co-designed architecture [4, 5, 13, 23, 24, 27]. However there is little work dealing with issues related to a multicore design, and none of the prior work investigate the potentially prohibitive challenge of shared data conflicts. Another domain of related work is that of deterministic replay on multicore processors [32, 22, 21]. These techniques leverage atomicity to implement efficient mechanism to guarantee replay of concurrent execution with deterministic interleaving of threads. The goal in these systems is to ”record” the shared data conflicts, so that the conflicts can later be deterministically replayed. These works do not propose how to *reduce* the shared data conflicts.

Finally, in the domain of transactional memory (TM) [17, 28, 16, 12], there have been significant efforts in investigating the cost of shared data conflicts and mitigating the costs. However, instead of programmers specifying atomic transactions, in a hybrid processor, the BT system decides atomicity boundaries and is free to choose and modify the boundaries. Further, a hybrid processor almost exclusively executes atomic regions. Because of these reasons, the cost of conflicts and our solutions are very different from those proposed in TM literature.

7 Conclusion

While hardware atomicity is the key enabling technology for achieving high performance in a hybrid architecture, when moving to a multicore design, this strength, can become a weakness. Frequent shared data conflicts across two atomic regions can result in expensive squashes and rollbacks. In this work, we have investigated how multithreaded applications, both benchmark and commercial workloads, are affected by these squashes at a range of both small and large atomic regions. While the current wisdom is that there are insignificant number of squashes for smaller atomic regions, we observe the opposite for many multithreaded workloads. With a region size of just 200 - 500 instructions, we observe a performance degradation ranging from 10% to more than 50% for workloads with a mixture of shared reads and writes. We have also presented a dynamic hybrid framework, **BlockChop**, for designing squash handling mechanisms to adaptively mitigate and eliminate squashes, and identified and evaluated 7 dynamic mechanisms using BlockChop. We find that using our chopper and adaptive chopper techniques over the state-of-the-art response to exceptions and squashes in a hybrid design, we are able to improve the performance of benchmark and commercial workloads by 1.4x and 1.2x on average for large and small region sizes, respectively.

References

[1] R. Agarwal and J. Torrellas. Flexbulk: intelligently forming atomic blocks in blocked-execution multiprocessors to minimize squashes. In *ISCA '11*, pages 33–44, New York, NY, USA, 2011. ACM.

[2] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. Bulkcompiler: high-performance sequential consistency through cooperative compiler and hardware support. In *MICRO 42*, pages 133–144, New York, NY, USA, 2009. ACM.

[3] C. Blundell, M. M. Martin, and T. F. Wenisch. Invisifence: performance-transparent memory ordering in conventional multiprocessors. In *ISCA '09*, pages 233–244, New York, NY, USA, 2009. ACM.

[4] E. Borin, Y. Wu, M. Breternitz, and C. Wang. Lar-cc: Large atomic regions with conditional commits. In *CGO '11*, pages 54 –63, april 2011.

[5] E. Borin, Y. Wu, C. Wang, W. Liu, M. Breternitz, Jr., S. Hu, E. Natanzon, S. Rotem, and R. Rosner. Tao: two-level atomicity for dynamic binary optimizations. In *CGO '10*, pages 12–21, New York, NY, USA, 2010. ACM.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, September 2008.

[7] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07*. Jun 2007.

[8] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: bulk enforcement of sequential consistency. In *ISCA '07*, pages 278–289, New York, NY, USA, 2007. ACM.

[9] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[10] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 97 –108, feb. 2007.

[11] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02*, pages 43 – 54, feb. 2002.

[12] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[13] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[14] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09*, pages 85–96, New York, NY, USA, 2009. ACM.

[15] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. In *Computers, IEEE Transactions on*, volume 50, pages 529–548. IEEE, 2001.

[16] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04*, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[18] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. Radbench: a concurrency bug benchmark suite. In *HotPar'11*, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[19] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *ASPLOS '10*, pages 77–90, New York, NY, USA, 2010. ACM.

[20] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08*, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

[21] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.

[22] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[23] N. Neelakantam, D. R. Ditzel, and C. Zilles. A real system evaluation of hardware atomicity for software speculation. In *ASPLOS '10*, ASPLOS '10, pages 29–38, New York, NY, USA, 2010. ACM.

[24] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07*, pages 174–185, New York, NY, USA, 2007. ACM.

[25] S. Patel and S. Lumetta. replay: A hardware framework for dynamic optimization. In *Computers, IEEE Transactions on*, volume 50, pages 590–608. IEEE, 2001.

[26] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *PACT '08*, pages 144–154, New York, NY, USA, 2008. ACM.

[27] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. In *ISCA '04*, pages 162–, Washington, DC, USA, 2004. IEEE Computer Society.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997. 10.1007/s004460050028.

[29] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The bulk multicore architecture for improved programmability. *Commun. ACM*, 52:58–65, December 2009.

[30] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. Smith, and M. Valero. Implementing kilo-instruction multiprocessors. In *ICPS '05*, pages 325 – 336, july 2005.

[31] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for store-wait-free multiprocessors. In *ISCA '07*, pages 266–277, New York, NY, USA, 2007. ACM.

[32] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 122 – 133, june 2003.