CHIP MULTIPROCESSORS WITH SPECULATIVE MULTITHREADING: DESIGN FOR PERFORMANCE AND ENERGY EFFICIENCY

BY

JOSE RENAU ARDEVOL

Ingen., Ramon Llull University, 1997
M.S., University of Illinois at Urbana-Champaign, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

Abstract

While Chip Multiprocessors (CMP) with Speculative Multithreading (SM) support have been gaining momentum, experienced processor designers in industry have reservations about their practical implementation.

SM CMPs must exploit multiple sources of speculative task-level parallelism, if they want to achieve enough performance improvement for non-numerical applications. Additionally, it is felt that SM is too energy-inefficient to compete against conventional superscalars.

This thesis challenges for the first time the commonly-held view that SM consumes excessive energy. It shows a CMP with SM support that is not only faster but also more energy efficient than a state-of-the-art wide-issue superscalar. This is demonstrated with a new energy-efficient CMP micro-architecture. To achieve these results, this thesis is also the first one to propose micro-architectural mechanisms that, taken together, fundamentally enable fast SM with out-of-order spawn in a CMP. These simple mechanisms are: Splitting Timestamp Intervals, the Immediate Successor List, and Dynamic Task Merging. To evaluate them, we develop a SM compiler with and without out-of-order spawn. In addition, the thesis identifies the sources of energy consumption in SM, and proposes energy-centric optimizations that mitigate them.

Experiments with the SpecInt 2000 codes show that a CMP with 4 3-issue cores and support for SM delivers a speedup of 1.27 over a 3-issue superscalar. The SM CMP is even faster than a 6-issue superscalar at the same frequency, and consumes only 85% of its power. In fact, for the same average power in both chips, the SM CMP is 1.13 times faster than the 6-issue superscalar on average.

To my family, friends, teachers, and Anna.

Acknowledgments

I want to thank my advisor Josep Torrellas for letting me work in the IACOMA group. I must give special thanks to James Tuck, Luis Ceze, Karin Strauss, Wei Liu, and Smruti Sarangi for their help to develop this thesis. Without them, my graduation would have been delayed and the whole process would have been more painful and boring. I want to thank all the other members in the IACOMA group (current and past): Paul Sack, Jun Nakano, Radu Teodorescu, Pablo Montesinos, Jose Martinez, Michael Huang, Milos Prvulovic, Yan Solihin. I also want to thank IBM for giving me a fellowship during the last year of my Ph.D.

More personally, I would like to thank Anna, my father, my sister, and all my other family members.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Substantial research effort is currently being devoted to speeding up hard-to-parallelize non-numerical applications such as SpecInt codes. Designers build sophisticated out-of-order processors, with carefully-tuned execution engines and memory subsystems. Unfortunately, these systems tend to combine high design complexity with diminishing performance returns, motivating the search for design alternatives.

One such alternative is Speculative Multithreading (SM) on a Chip Multiprocessor (CMP) [8, 15, 16, 19, 23, 31, 32, 36, 37, 46]. Under SM, irregular sequential codes are divided into tasks that are executed in parallel, optimistically assuming that sequential semantics will not be violated. As the tasks run, the architecture tracks their control flow and data accesses. If a cross-task dependence is violated, the offending tasks are destroyed (*squashed*). Then, a repair action is initiated and the offending tasks are re-executed.

While these architectures have shown good potential, often due to sophisticated compiler support [4, 9, 21, 38, 40, 44], the speedups obtained for non-numerical applications have typically been modest. For example, for full SpecInt 2000 applications, the geometric mean speedups are 1.05 [44].

Besides performance considerations, processor designers in industry have reservations about the practical implementation of SM. In particular, it is felt that SM is too energy-inefficient to seriously challenge superscalars. The rationale is that aggressive speculative execution of possibly incorrect tasks is not the best course at a time when processors are primarily constrained by energy issues. Clearly, for SM CMPs to even be considered, their energy and power requirements must be competitive with wide-issue superscalars.

This thesis addresses *Energy* and *Performance* issues in SM. First, it proposes a hardware and

a compiler that boosts the performance for non-numerical codes (SpecInt) through out-of-order task spawn. Second, it identifies the specific sources of energy consumption in SM, and proposes energy-centric optimizations that directly target them.

1.1 Performance

The performance achieved by previously proposed SM CMPs for non-numerical applications has been modest. Part of the reason is that most designs have typically focused (often implicitly) on limited types of task structures: iterations from a single loop level (e.g. [8, 19, 44]); the code that follows (i.e., the continuation of) calls to subroutines that do not spawn other tasks (e.g. [7]); or some execution paths out of the current task (e.g. [40]). In the cases mentioned, *correct* tasks are spawned *in-order*, namely in the same order as they would execute sequentially[1]. While exploiting only these task structures may simplify the CMP hardware, it cripples the potential of SM. High-level performance evaluation studies for SM have pointed out that there is a sizable amount of parallelism available [24, 26, 42, 43].

One could execute in parallel all subroutines and their continuations irrespective of their nesting, and iterations from multiple loop levels in a nest. If this additional parallelism is harvested, the speedups are predicted to be significantly higher. In practice, exploiting these additional sources of parallelism requires supporting *out-of-order* task spawning. For example, consider nested subroutines. When a task finds a subroutine call, it spawns a more speculative task to execute the continuation, while it proceeds to execute the subroutine. Inside the subroutine, the task can then find other subroutine calls, therefore spawning speculative tasks that are less speculative (i.e., less ahead in a sequential execution) than the one spawned first. The same occurs for nested loops, and for combinations of loop and subroutine nesting.

With out-of-order spawning, the application offers unpredictable shapes of parallelism that are hard to manage by SM at run time. Specifically, how to manage task ordering, which is required to identify violations and to ensure correct commit and squash? How to balance resource allocation between highly-speculative tasks that have been running for a long time, and less speculative tasks that have just been spawned? Special microarchitecture support is needed to address these

[1]Correct tasks do not include those that are in wrong branch paths, which ultimately get squashed.

challenges with *minimal overhead* in a CMP.

The concept of out-of-order spawn is not new. In fact, there is a lot of related work in this area, which is detailed in Section 9. For example, Hammond *et al.* [16] used software to control an environment with out-of-order spawning (their findings motivate our work). Several authors performed high-level performance evaluation analyses of environments with out-of-order spawning, typically simulating simplified architectures [24, 26, 42, 43]. One paper presented the microarchitecture for out-of-order spawn in the DMT *centralized* multithreaded core [2]. Finally, other authors presented mechanisms that could be adapted to help speed up out-of-order spawning [12, 13, 31].

However, no previous work has proposed a set of implementable microarchitectural mechanisms that, altogether, fundamentally enable high-speed tasking with out-of-order spawn in a SM CMP. This thesis is the first to do it.

The simple mechanisms address the two main challenges posed by out-of-order spawning: correct and efficient task ordering and resource allocation. Task ordering is enabled with *Splitting Timestamp Intervals* for low-overhead order management, and the *Immediate Successor List* for efficient task commit and squash. Efficient resource allocation is enabled with *Dynamic Task Merging*, which directs speculative parallelism to the most beneficial code sections.

1.2 Energy

Power issues have become the main concern for designers of high-end microprocessors. Energy and power consumption directly affect the cost of powering and cooling the system, influence the reliability and aging characteristics of chips, and determine battery life in portable devices. While the simpler cores in a CMP are energy-efficient, CMPs with SM will not be accepted unless their overall energy requirements are competitive against wide-issue superscalars.

This thesis also addresses the problem of energy consumption in SM. It shows that, perhaps contrary to commonly-held views, SM *does not* excessively consume energy. This is the first work to show that a SM CMP can be a very competitive design for high-performance, power-constrained processors, even under the very challenging SpecInt workloads.

Fundamentally, the energy cost of SM can be kept modest by using a lean SM CMP microarchitecture and by minimizing wasted SM work. Then, such an energy-efficient SM CMP can beat

a wider-issue superscalar simply because, as the size of the processor structures increases, energy scales superlinearly and performance sublinearly.

After identifying the main sources of energy consumption in SM, we propose energy-centric optimizations that directly target them. These sources are the wasted work of squashed tasks, storage and logic in the memory hierarchy to support data versioning, additional traffic in the memory subsystem, and additional instructions due to hard-to-optimize code.

1.3 Contributions and Main Results

This thesis offers two complementary sets of contributions in performance and energy. That when taken together show that a SM CMP is not only faster but also more energy-efficient than a state-of-the-art wide-issue superscalar.

Energy: Enable energy-efficient designs with SM.

- Identify and analyze the main sources of energy consumption in SM. These sources are the wasted work of squashed tasks, storage and logic in the memory hierarchy to support data versioning, additional traffic in the memory subsystem, and additional instructions due to hard-to-optimize code
- Propose energy-centric optimizations that mitigate the SM sources of energy consumption. These optimizations have been overlooked in performance-centric SM designs because they enhance energy-savings and not performance.
- Design a new energy-efficient memory hierarchy for a CMP with SM.

Performance: Novel micro-architectural mechanisms to enable out-of-order spawn in a SM CMP.

- *Splitting Timestamp Intervals* for task ordering with low-overhead order management.
- *Immediate Successor List* for efficient task commit and squash token propagation.
- *Dynamic Task Merging* for efficient resource allocation.

This is the first time that SM on a CMP is an interesting design point even for high-performance power-constrained designs. To do so, we have developed a complete, fully-automated SM compiler

for aggressive out-of-order spawn. With the SM CMP architecture proposed, a SM CMP with 4 3-issue cores delivers an average speedup of 1.27 for *full* SpecInt 2000 applications; without out-of-order spawn, the average speedup is 1.05, in line with past SM CMP work on the same codes (e.g., 1.05 in [44]). Moreover, the resulting CMP significantly outperforms a 6-issue superscalar, even at the same clock frequency. Overall, the mechanisms to enable out-of-order spawn unlock the potential of SM for the toughest applications, namely irregular integer codes.

Once the performance desired is achieved, the proposed energy-centric optimizations cut *the energy cost of SM* by half. In global terms, they eliminate 20% of the energy in the SM CMP without impacting performance. The result is a SM CMP faster than a 6-issue superscalar at the same frequency, while consuming 85% of its average power. In fact, for the same average power in both chips, the SM CMP is 13% faster than the wider superscalar for these most challenging codes. We expect much better results for floating point, multimedia, or more parallel codes.

The thesis is organized as follows: Chapter 2 provides background on SM; Chapter 3 provides an introduction to out-of-order spawn; Chapter 4 analyzes the sources of energy consumption in SM; Chapter 5 describes the proposed SM CMP architecture; Chaper 6 describes our SM compilation infrastructure; Chapters 7 and 8 present our evaluation methodology and the evaluation; Chapter 9 describes the related work; and Chapter 10 concludes.

Chapter 2

Background Information

Speculative Multithread (SM) is also known as Thread-Level Speculation (TLS) or Speculative Parallelization. In all the cases, it extracts tasks from a sequential code and speculatively executes them in parallel, hoping not to violate sequential semantics. The control flow of the sequential code imposes a task order and, therefore, we use the terms predecessor and successor tasks. The safe (or non-speculative) task precedes all speculative tasks. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks, which parallel execution cannot violate. As tasks execute, special hardware support checks that no cross-task dependence is violated. If any is, the incorrect tasks are squashed, any polluted state is repaired, and the tasks are re-executed.

2.1 Cross-Task Dependence Violations

Data dependences are typically monitored by tracking, for each individual task, the data written and the data read with exposed reads. The dependence can be tracked at word or a cache line level. The former has a granularity of 32 or 64 bits, while the latter has a granularity equal to the cache line size.In all the cases. A *write* always marks the cache line as dirty. If the datum size is equal to the granularity, the protecting write bit for that datum is set. For example, in a word base granularity, a write to a word sets the protecting write bit for that word. Note that even a word granularity would not set the protecting write bit only if a byte is stored. A *read* marks the exposed read bit for that datum unless a protecting write bit is set for the same datum. A data dependence violation occurs when a task writes a location that has been read by a successor task

with an exposed read.

A control dependence violation occurs when a task is spawned in a mispredicted branch path. Dependence violations lead to task squashes, which involve discarding the work produced by the task. Squashes come in two forms. In a control violation, the task is squashed with kill signal. In a data violation, the task is squashed with a restart signal, which also restarts the task from its beginning, hoping that the re-execution will not violate the data dependence.

2.2 State Buffering

Memory accesses issued by a speculative task must be handled carefully. Stores generate speculative state that cannot be merged with the safe state of the program. The reason is that it may be incorrect. Consequently, the state is stored separately, typically in the cache of the processor running the task. If a violation is detected, the state generated by the task is discarded. Otherwise, when the task becomes non-speculative, the state is allowed to propagate to memory. When a non-speculative task finishes execution, it commits. Committing informs the rest of the system that the state generated by the task is now part of the safe program state. Commit is done in task order and involves passing a commit token between tasks.

2.3 Data Versioning

A task has at most a single version, also known as timestamp, of any given variable. However, different speculative tasks that run concurrently in the machine may write to the same variable and, as a result, produce different versions of the variable. Such versions or timestamps must be buffered separately. Moreover, when a speculative task reads, it needs to be provided with the closest predecessor version of the variable. Finally, as tasks commit in order, data versions need to be merged with the safe memory state also in order.

2.4 Multi-Versioned Caches

A cache that can hold state from multiple tasks is called multi-versioned [10, 15, 32]. There are two performance reasons why multi-versioned caches are desirable: they avoid processor stall when

tasks are imbalanced, and enable lazy commit.

If tasks have load imbalance, a processor may finish a task and the task still be speculative. If the cache can only hold state for a single task, the processor has to stall until the task becomes safe. An alternative is to move the task's state to some other buffer, but this complicates the design. Instead, it is best that the cache retain the state from the old task and allow the processor to execute another task. If so, the cache has to be multi-versioned.

Lazy commit [28] is an approach where, when a task commits, it does not eagerly merge its cache state with main memory through ownership requests [32] or write backs [19]. Instead, the task simply passes the commit token to its successor. Its state remains in the cache and is lazily merged with main memory later, usually as a result of cache line replacements. This approach improves performance because it speeds up the commit operation. However, it requires multi-versioned caches.

Tagging Multi-Versioned Caches. Multi-versioned caches typically require that we tag each cache line with a version ID, which records what task the line belongs to. Intuitively, such version ID could be the task version. Unfortunately, the ID of a task can be quite long. Consequently, to save space, it is best to translate the version into some arbitrary Local IDs (LIDs) that are much shorter. These LIDs are used only locally in the cache, to tag cache lines. This type of ID indirection was first used by Steffan *et al.* [34].

While these LIDs save space in the tags, they need to be translated. They can be kept in a small, per-cache table that we call LID Table. Each cache has a different LID Table.

2.5 Architecture and Environment Considered

SM can be supported in different ways. This thesis focuses on a Chip Multiprocessor (CMP) architecture because it is a decentralized, potentially energy-efficient platform. To reduce non-commodity hardware, we assume that the processors in the CMP can only communicate via the memory system — there is no hardware support for register communication. In addition, to gain flexibility, the speculative tasks are generated in software by a SM compiler. This is a new compiler that we recently built. Finally, we concentrate on SpecInt 2000 applications, as these non-numerical applications are hard to speed up with conventional platforms.

Chapter 3

Out-of-Order Spawn Model

In most of the proposed SM systems, tasks are formed with iterations from a single loop level (e.g. [8, 19, 44]), the code that follows (i.e. the continuation of) calls to subroutines that do not spawn other tasks (e.g. [7]), or some execution paths out of the current task (e.g. [40]). In these proposals, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution path of the program. As a result, tasks are spawned *in order*, namely in the same order as they would in sequential execution.

Figures 3.1-(a) and (b) show examples. Figure 3.1-(a) shows the task tree when parallelizing a loop. Each task spawns the next iteration. In the figure, the leftmost task is safe (or non-speculative); the more a task is to the right, the more speculative it is. Figure 3.1-(b) shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while a more speculative task is spawned to execute the continuation.

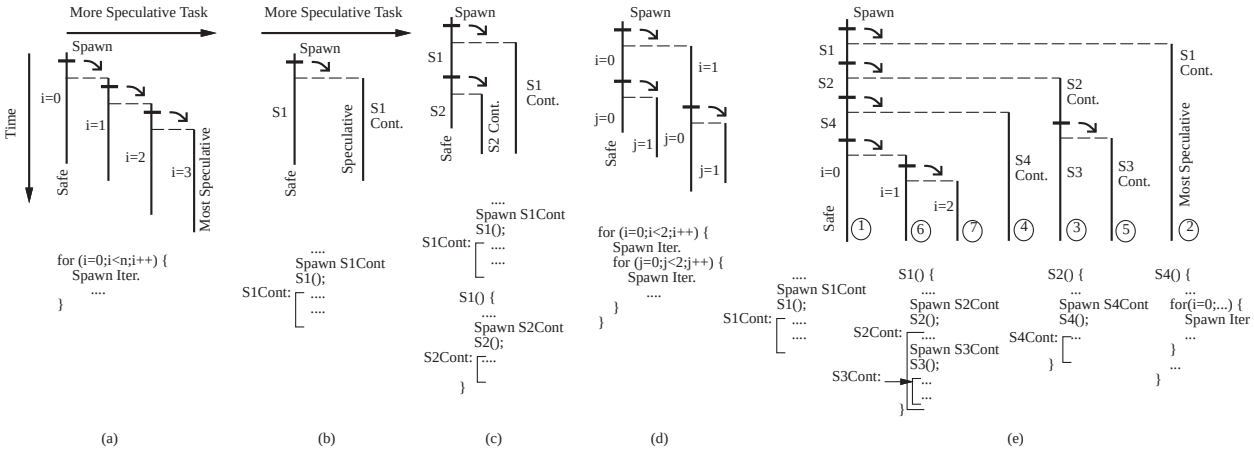

Figure 3.1: Task trees resulting from different approaches to build SM tasks. In the figure, Cont and Iter denote continuation and iteration.

There is consensus that for SM to deliver on its promise, it has to exploit more parallelism. Several high-level performance evaluation studies [24, 26, 42, 43], typically simulating simplified architectures, have pointed to the need to support nested subroutines and loop iterations.

Figures 3.1-(c) and (d) show the two cases. In Figure 3.1-(c), the safe task first spawns a task for the continuation of subroutine *S1*. Then, it enters *S1*, spawns a new task for the continuation of *S2*, and executes *S2* until its end. In Figure 3.1-(d), the safe task executes outer iteration 0. As it executes, it spawns outer iteration 1, enters the inner loop to execute inner iteration 0, and spawns inner iteration 1. When it completes inner iteration 0, it ends.

With these two task choices, an individual task can spawn multiple correct tasks. If so, correct tasks are spawned in strict reverse order compared to sequential execution. For example, in Figures 3.1-(c) and (d), the safe task spawns two correct tasks, and does so out of order, most speculative first. Figure 3.1-(e) is a more complex example: the time-line for task creation proceeds from top to bottom (*1-2-3-4-5-6-7*), while sequential order is from left to right (*1-6-7-4-3-5-2*).

This thesis, to discuss out-of-order spawning, gives examples of tasks built out of any nesting of subroutines and loop iterations, as they are an obvious source of SM parallelism. The analysis also applies to any other task structure that maintains two conventions. First, if a task spawns multiple tasks, the compiler inserts the spawns in strict reverse task order (last task is spawned first, etc). Second, the spawned tasks are less speculative than any task that was more speculative than their parent. These conventions are followed to make the spawn structure like that of nested loops and subroutines. Intuitively, these conventions are unlikely to affect task selection much, while they simplify the microarchitecture.

3.1 Why Supporting Out-of-Order Spawning?

Out-of-order spawning enables more task parallelism: two code sections that are far-off in sequential execution can be executed in parallel *before* some of their intervening code sections have *even been spawned*.

As a simple example, consider Figure 3.1-(c) when a task (say, *T1*) reaches the call to *S1*. Under out-of-order spawn, it can immediately spawn *S1*'s continuation. Under in-order spawn, if we want to achieve the same degree of parallelism, *T1* has to instead spawn, right there, its immediate

successor task (since *T1* can only spawn a single task ever). In the example, the immediate successor is the continuation of *S2*. In general, the immediate successor may be deeply nested inside the code. Identifying the successor may require run-time computation to determine the true control flow. This requires executing overhead instructions. The alternative is for *T1* to continue executing without spawning, until it is clear that the *S2* continuation is its immediate successor task. In either case, in-order spawn loses parallelism.

3.2 Why Supporting Out-of-Order Spawn in CMPs is Hard?

Generally, with out-of-order spawn, *all* tasks can spawn, and parallelism expands in unexpected parts of the task tree at run time. As a result, in decentralized architectures such as CMPs, it becomes harder to maintain two cornerstones of SM: task ordering and efficient resource allocation.

Task ordering is required in several SM operations that are time-critical. Specifically, a task needs to know its immediate successor, to communicate the commit token or a squash signal. Moreover, any communication between two tasks requires knowing the tasks' relative order: such order determines whether a dependence violation is triggered, or what data version is returned to the requester. Unfortunately, when fine-grain tasks are spawned out of order, unpredictably and in different processors, high-speed ordering of tasks and its maintenance is hard.

Efficient allocation of resources (e.g. CPU or cache space) is crucial for SM performance. Ideally, resources should be assigned to tasks that are safe or very likely to become so. However, with out-of-order spawning, there may be highly-speculative tasks that have been running for a long time. In this case, if the safe task wants to spawn and there are no free CPUs, should it kill the highly-speculative tasks? This is what past schemes do [16, 24]. Or should it abstain from spawning, do the work itself, and leave the highly-speculative tasks running?

Since decisions on task ordering and resource allocation have to be made very quickly, they need to be supported in the CMP micro-architecture. Given the complexity of SM designs, however, such new micro-architecture needs to be simple.

3.3 Out-of-Order Spawning and Number of Processors

With out-of-order spawning, SM can unlock additional parallelism: two code sections that are very separated in sequential execution can be executed *before* some of their intervening code sections have *even been spawned*. This feature enables more task overlap, and can benefit both machines with many processing elements (PE) and those with only few.

On the other hand, it is well-known that some integer applications have only modest coarse-grained parallelism. For example, for SpecInt, few-PE machines have often been a sweet spot. For these applications, even with out-of-order spawning, it is reasonable to target two-PE machines. Indeed, in code sections where, without out-of-order spawning, one of the two PEs of the machine would remain idle, we may now overlap the execution of two tasks that are far apart in sequential execution.

Chapter 4

SM-Specific Sources of Energy Consumption

Enhancing a superscalar into a CMP with SM support causes the energy consumption to increase.

This chapter identifies the SM-specific sources of dynamic energy increase and propose optimizations to mitigate them. This chapter only analyzes the sources, the proposal of energy-centric optimizations that directly mitigate them is shown in Section 5.3.

The main SM-specific sources of dynamic energy consumption are shown in Table 4.1. They are task squashing, storage and logic for data versioning in the memory hierarchy, additional traffic in the memory subsystem, and additional instructions due to hard-to-optimize code. These sources are analyzed next.

Another source of dynamic energy overhead is the replication of the cores on chip. These cores may not be busy all the time. However, we use the well-known technique of clock gating to eliminate most of the energy waste when the cores are unused. We do not consider this source as SM-specific.

SM-Specific Sources of Energy Consumption
Task squashing
Storage and logic for data versioning in the memory hierarchy
Additional traffic in the memory subsystem
Additional instructions (in non-squashed tasks) due to hard-to-optimize code

Table 4.1: Main SM-specific sources of energy consumption.

4.1 Task Squashing

A source of energy consumption specific to SM is the work performed by tasks that ultimately get squashed. Note, however, that not all such work is necessarily wasted. Specifically, in a data dependence violation, a task is squashed and often restarted on the same processor. The new instance of the task can leverage branch prediction training from the previous instance. More importantly, the hardware should allow the new instance to reuse the non-dirty state left in the cache by the previous instance. As a result, the new execution can be faster than the previous one.

Task squashing also consumes energy in two other operations: sending the squash signal to the processor where the incorrect task is running, and possibly executing some re-initialization code on that processor. Such code may involve restoring the register state, but does not require accessing any large chunk of data in the caches. Given the low frequency of squashes, the energy consumed in these two operations is negligible.

4.2 Storage and Logic for Data Versioning in the Memory Hierarchy

In SM, since different tasks may be accessing and updating the same address concurrently, the memory hierarchy may have to hold multiple versions of the same datum. The resulting additional storage and logic required to support data versioning is a SM-specific source of energy consumption.

In many proposed SM schemes, individual caches are multi-versioned (e.g. [10, 15, 32]), which means that they can hold state from multiple tasks (Section 2). In this case, each cache line is typically tagged with a short version ID, which identifies the task it belongs to. Moreover, in many systems, messages between caches also include the requesting task's ID. On an external access to a cache, the version ID of an address-matching line in the cache is compared to the ID in the incoming message. From the comparison, the cache may determine that a violation occurred. Overall, supporting data versioning can require extra storage in caches and messages to hold version IDs, and extra logic to compare those IDs when communication occurs. The details of the memory hierarchy that we use are shown in Section 5.

4.3 Additional Traffic in the Memory Subsystem

A SM system generates a higher number of messages than a superscalar. While some of these extra messages are the result of parallel execution, there are three more SM-specific reasons for the increased message volume.

One reason is that caches do not work as well. Caches often have to retain lines from older tasks that ran on the processor and are still speculative. Only when such tasks become safe can the data be displaced. As a result, there is less space in the cache for data that may be useful to the task currently running locally. This higher cache pressure increases displacements of useful lines and subsequent misses.

The presence of multiple versions of the same line in the system also causes additional messages. Specifically, when a processor requests a line, multiple versions of it may be provided, and the processor (or the directory) then selects what version to use.

Finally, it is desirable that the speculative cache coherence protocol track dependences at a fine grain, which creates additional traffic. To see why, recall how these protocols typically track dependences: they record which data are written and which data are exposed-read in each task (Section 2). This information is often encoded with a Write (W) and an Exposed-Read (R) bit per cached datum.

If this access information is kept per line (Figure 4.1-(a)), tasks that exhibit false sharing may appear to violate data dependences and, as a result, cause squashes [10]. For this reason, many SM proposals keep some access information at a finer grain, such as per word (Figure 4.1-(b)). Unfortunately, per-word dependence tracking induces higher traffic: a message (such as an invalidation) may need to be sent for each and every word of the line.

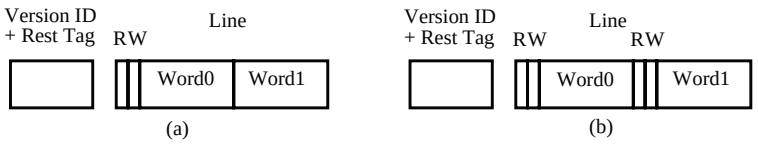

Figure 4.1: Keeping access information per line (a) or per word (b).

4.4 Additional Instructions

SM systems with compiler-generated tasks often execute more dynamic instructions than non-SM systems. There are two sources of these additional instructions: side-effects of breaking the code into tasks and, less importantly, SM-specific operations.

The majority of additional instructions result from two side-effects of task generation. First, conventional compiler optimizations are not very effective at optimizing code across task boundaries. Therefore, code quality is relatively lower. Secondly, in CMPs where processors communicate only through memory, the compiler often spills too many registers across task boundaries.

SM-specific operations are the other source of additional instructions. They include task spawn and commit instructions. The spawn instruction involves sending some state from one processor to another. In our implementation, this state is the program counter, the stack pointer, and a handful of other values. In other implementations, it may also involve executing a few instructions in the sender or receiver. Efficient, lazy implementations of task commit consist in sending the commit token from one processor to another [28] (Section 2). They do not involve any significant transfer of data or messages in the system: committed data are later transferred to memory on cache replacements. This is the approach that we use. In other implementations, commit may also involve executing a few instructions in both processors [14]. Overall, given the modest frequency of spawns and commits, their combined energy is very small.

Chapter 5

Architecture Description

Based on the previous discussions in Chapters 3 and 4. This chapter outlines the CMP with SM architecture proposed.

In the following, the hardware structures and then the functionality are explained as follows: Section 5.1 explains a version memory for energy-efficiency; Section 5.2 shows how the speculative multithreading protocol; Section 5.3 proposes novel energy-centric optimizations; Section 5.4 introduces a novel Timestamp Intervals to support task order management when tasks are spawned out-of-oder; Section 5.5 explains a immediate successor list to support squash and commit; and finally Section 5.6 explains a novel dynamic task merging for efficient resource allocation.

5.1 Version Memory for Energy-Efficiency

The proposed architecture is a small-scale CMP with two (or four) modest-issue processors. Each processor has a private, multi-versioned L1. All the processors share a small, multi-versioned victim cache that holds lines overflowing from the L1s. Finally, there is an unmodified, shared L2 that only holds safe data (Figure 5.1).

The interconnect between the L1 caches and the victim cache is a switch. We include a victim cache to avoid the much more expensive alternative of designing a multi-versioned L2. The combined space of the L1s plus the victim cache is practically always sufficient to hold the speculative state of all the running tasks; only rarely does a task get squashed due to lack of cache space. Figure 5.2 shows the extensions performed on the processors. Each structure shows its fields in the form bit_count:field_name.

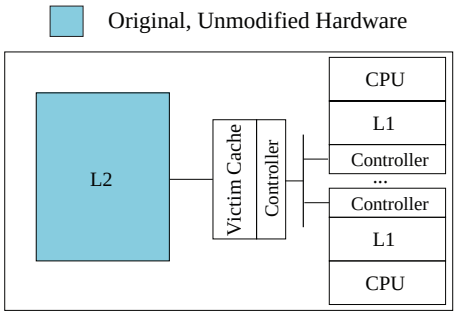

Figure 5.1: Proposed speculative multithreading chip multiprocessor architecture (SM CMP).

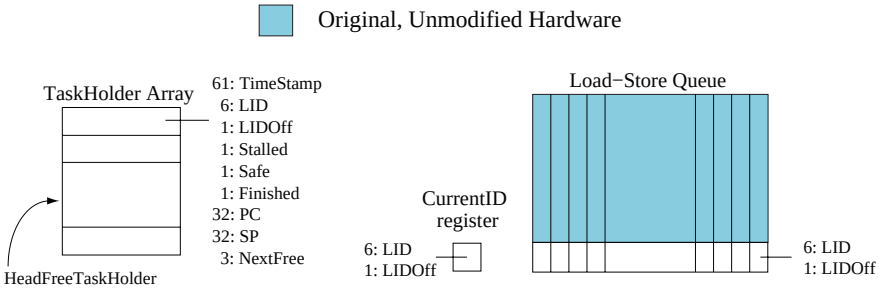

Figure 5.2: Proposed processor modifications to support Speculative Multithreading.

Each processor has an array of *TaskHolders*, which are hardware structures that hold some minimum state for the tasks that are currently loaded on the processor (Figure 5.2). Each TaskHolder contains the task's LID, LIDOff (incremented every time the task causes a violation, as per Section 5.3.2), a Stalled bit (set when the task causes a second violation and is forced to stall as per Section 5.3.1), Safe and Finished bits (set when the task receives the commit token and finishes execution, respectively), the task spawn address (PC), its stack pointer (SP), a pointer to the next free TaskHolder, a counter called *Number of Ends to Skip* (NES) explained in section 5.6.1, Timestamps are explained in section 5.4, and a pointer to the *Immediate Successor* (IS) explained in section 5.5. The TaskHolder does not store the register state, which is kept in the stack. A TaskHolder can be recycled when the owner task has committed and passed the commit token to its successor.

The table of TaskHolders is accessed by instructions such as spawn, and hardware signals such as restart or kill. Consider, for example, the case when a task must kill all its successors. In this case, the hardware passes the kill signal from the originating task down the IS list. For each task in the list, the operation is as follows. If the task is running, it is stopped and the pipeline is flushed.

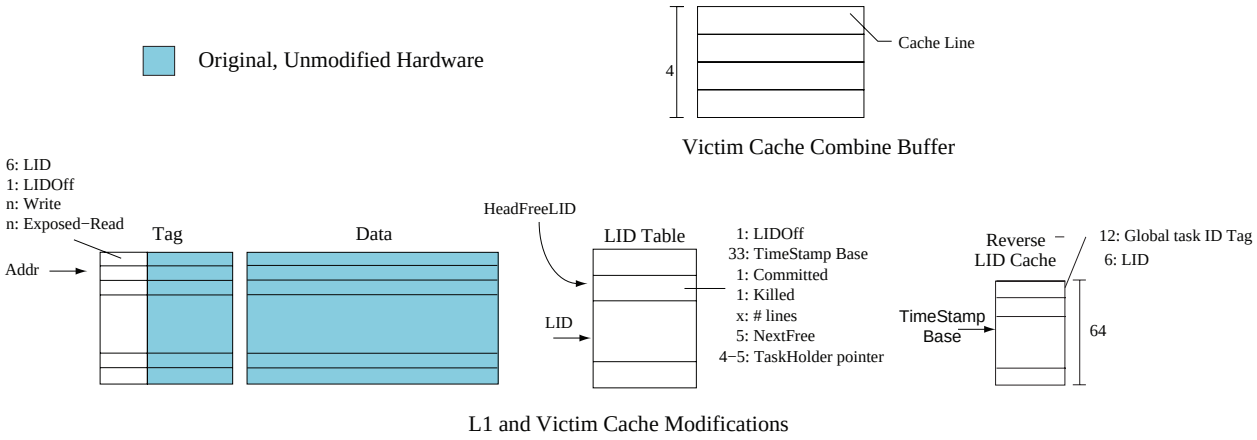

Figure 5.3: Proposed memory hierarchy modifications to support SM CMP. n is the number of words per cache line, x is $\log_2$(Cache Size).

In all cases, the task's LID is marked as invalid, so that the task's cache lines become invalid and can be purged lazily. As in typical SM systems, that LID remains unused until all its lines are purged from the cache; at that point, it can be reused. If a task needs to be restarted, the initial PC and stack pointer are restored from the task context, and the LIDOff is incremented.

A copy of the LID and LIDOff for the task currently going through rename is kept in the *CurrentID* register of the load-store queue (Figure 5.2). Such register is used to tag loads and stores as they are inserted in the load-store queue. With this support, the processor can have multiple in-flight tasks, and all accesses to the multi-versioned caches carry with them the correct LID and LIDOff.

Figure 5.3 shows the extensions required by SM to L1 caches and victim cache. In the L1s and the victim cache, each line tag is augmented with LID, LIDOff and, for each word in the line, one Write and one Exposed-Read bit to record accesses (Figure 5.3). As per Section 2, each cache keeps its own LID to version translations in a LID Table (Figure 5.3). The LID Table is direct mapped and is indexed by a LID. Each entry has information for one LID: its corresponding version, LIDOff, bits that indicate if the task is killed or committed, a counter of the number of lines in the cache belonging to that LID (Section 5.3.2), a global pointer to the corresponding TaskHolder, and a pointer to the next free entry. An entry can be recycled when its line counter is zero and the TaskHolder it points to has been recycled.

LIDs are local per cache. Since lines passed between caches include the version, each cache

needs a small *Reverse LID Cache* to translate from version to own LID (Figure 5.3). If an access to the Reverse LID Cache misses, the LID Table is traversed, and a new entry is allocated in the Reverse LID Cache.

Finally, the victim cache has a *Combine Buffer* that is used when a safe line is about to be displaced to L2 (Figure 5.3). The buffer first requests from the caches all the safe versions of that line. As they arrive, the buffer combines them, so that each word in the line has the latest safe version on chip. Then, the line is committed to L2, and the other versions invalidated.

5.2 Speculative Multithreading Protocol

To operate these structures the cache coherence protocol is extended to support speculation. To see how these structures are used, we now describe a task spawn, load hit and miss in L1, line displacement to L2, and task restart.

Task Spawn

When a processor executes a spawn instruction, it sends a small packet with the starting PC, SP, and version to another processor. In the latter, the hardware allocates a new TaskHolder and initializes it as follows: LID is set to the value pointed to by HeadFreeLID (Figure 5.3); LIDOff, Stalled, Safe, and Finished are reset; and PC and SP are set to the values received in the message. The fields in the corresponding LID Table entry are also initialized. The version is set with information from the message, the LIDOff, Committed, Killed, and line counter fields are reset, and the TaskHolder pointer is set to point to the TaskHolder. At this point, the task is ready to execute.

When the first instruction of a task goes through the rename stage, the LID and LIDOff from the TaskHolder are copied to the CurrentID register in the load-store queue. At any time, when an entry in the load-store queue is filled, it is also tagged with the CurrentID. As a result, when a load or store request is issued to L1, it carries with it the task's LID and LIDOff.

Load Hit/Miss in L1

If a load's address, LID, and LIDOff match one of the L1 tag entries, a hit is recorded and the data is returned immediately. If, instead, both address and LID match but LIDOff mismatches, the Write bits in the line are checked. If all of them are zero, the data is also returned as in a hit, and the line is promoted by setting the tag's LIDOff to the request's LIDOff (Section 5.3.2). This

operation also clears the Exposed-Read bits except for the loaded word. Note that the LID Table is *not* accessed in either case.

In all other cases, a miss is recorded and the LID Table is accessed. We index the LID Table with the request's LID, obtain the corresponding version, and include the latter in a request issued. Moreover, to decide which line to displace from L1, we also index the LID Table with the LIDs of the lines that are currently using the L1 set where space is needed. All these non-critical accesses proceed as fast as the number of read ports in the LID Table would allow. With the information retrieved from the LID Table, we can select the victim line — for example, one whose LID Table entry has the Killed bit set. If the victim line has to be displaced to another cache, the victim's LID Table entry provides the version to include in the message. In all cases, the victim's LID Table entry is updated by decrementing its count of lines in L1.

A miss request will reach all the other L1 caches and the victim cache. When a cache receives the request, it checks if it has a version of the line. It is possible that several lines match the address. The LIDs of these lines are used to index the local LID Table and retrieve the corresponding version to assess their relative order. The cache will assemble the lastest local version of the line that still precedes the requester task [28]. If any such line is generated, it is combined with the request.

This process is repeated by all caches and the victim cache which, all together, end up assembling the lastest version of the line on chip. After the victim cache completes its operation, the line is forwarded to the requester. If no matching line was found on chip, the victim cache initiates a read to L2. Therefore, the L2 cache is not accessed, until all the L1 caches are checked.

Line Displacement

When a cache needs space, it can displace a committed or uncommitted line by displacing it on the victim cache. If the victim cache can not absorve the displaced cache line, the most speculative task in the set of the victim cache would get a restart. For the victim cache to accept an incoming line, it needs to know the local LID that corresponds to the line's version. Such LID is needed to insert the line in the receiving cache. To find this LID, the version is used to index the Reverse LID Cache of the receiving cache. If a matching entry is found, it returns the LID. Otherwise, the LID Table is traversed to find out if a local translation exists. Note that this operation is not time critical. If no translation exists, a new one is created. The cache can now attempt to absorb the

line. However, if all the entries in the target set are used, the cache makes room by shedding a committed line or a line more speculative than the incoming one. If no room can be found, the incoming line is rejected.

If, after the victim cache completes its operation there is still an uncommitted line that cannot be absorbed, the task that owns it is restarted. If the line that cannot be absorbed is safe, the victim cache uses the Combine Buffer to send it to L2 (Section 5.1).

Task Restart

When a cache detects a violation, the TaskHolder of the task that performed the stale read is examined. If the task is running, it is stopped. Its state is then re-started and its LIDOff in the TaskHolder is incremented. In addition, a message is broadcast on all the LID Tables to increment the LIDOff for the task. If the task gets a second restart, LIDOff is 2, the Stalled bit in the TaskHolder is set and the task is stalled until it becomes safe. Otherwise, the task is allowed to re-execute.

Scheduling Tasks to CPUs

While all the tasks that have been spawned have their state loaded on on-chip task contexts, only as many tasks as CPUs can be running at a time. In practically all SM proposals, tasks are scheduled strictly based on how speculative they are. Specifically, a less speculative task always preempts more speculative ones. Moreover, among the eligible tasks, the preempted one is the most speculative.

In practice, our evaluation will show that such a policy is an overkill, given the typical load and task sizes in our CMP, and our new task merging support. Consequently, we propose and use a simpler policy: we assign high priority to the non-speculative task, and a fixed low priority to all speculative tasks. There are no complex priorities and only the safe task can preempt.

5.3 Novel Energy-Centric Optimizations

To reduce the energy cost of Speculative Multithreading, we could use many performance-oriented SM optimizations proposed elsewhere. Examples are mechanisms to reduce the number of squashes [11, 33] or improvements to the speculative coherence protocol [32]. While these optimizations improve performance, they typically also reduce the energy consumed by a program.

This thesis do not evaluate previously proposed optimizations. If they are effective for performance, we include them in our SM design hoping that they would also reduce energy consumption. Additionally, we are interested in "energy-centric" optimizations. These are optimizations that do not increase performance noticeably; in fact, they may slightly reduce it. However, they reduce energy significantly. We focus on these optimizations because they have been traditionally overlooked in performance-centric SM designs.

The specific sources of energy consumption in Speculative Multithreading are explained in Section 4. Table 5.1 extends Table 4.1 by summarizing the energy-centric optimizations proposed.

SM-Specific Sources of Energy Consumption	Proposed Energy-Centric Optimizations
Task squashing	Stall a task after its second restart
	Energy-aware task pruning by profiling
Storage and logic for data versioning in the memory hierarchy	Avoid eagerly "walking" the cache tags
Additional traffic in the memory subsystem	
Additional instructions (in non-squashed tasks) due to hard-to-optimize code	Energy-aware task pruning by profiling

Table 5.1: Main SM-specific sources of energy consumption and energy-centric optimizations to mitigate them.

5.3.1 Task Squashing

We propose two optimizations to reduce energy consumption due to task squashing.

1. Stall a Task After Its Second Restart. When a task that has caused a data dependence violation and has been restarted already once causes a second violation, we propose to stall it for good. The task is not given a CPU again until it becomes non-speculative. This optimization is energy-centric: a performance-only approach would keep re-executing the task with the hope that one of the runs completes without violations.

Note that, when a task receives its first restart signal, we re-execute it immediately. We do this hoping to reuse the state in the branch predictor and caches. Often, the first data dependence violation is due to the live-ins between parent and child and, after restart, no more violations will occur. A second violation may indicate the existence of too many true dependences to make speculative execution worthwhile. Consequently, we stall the task.

2. Energy-Aware Task Pruning by Profiling. Careful task pruning by a profiler pass attempts to identify and retain "beneficial squashes". Specifically, we propose an energy-centric profiler that attempts to minimize the product $Energy \times Delay^2$ for the program. Any task execution and subsequent squash that decreases the product is allowed; those that increase it are disallowed, since voltage-frequency scaling can (ideally) do better.

Our SM compiler generates a binary with spawn instructions to start tasks at run time (Figure 5.4-(a)). The binary is passed to a profiling pass, which executes it sequentially, using a profiling input (Section 6.3). The profiler estimates if a task squash will occur and, if so, the number of instructions squashed $I_{squashed}$ (Figure 5.4-(b)) and the final instruction overlap after re-execution $I_{overlap}$ (Figure 5.4-(c)). In addition, the profiler estimates the number of misses in the machine's L2 cache for the squashed instructions $M_{squashed}$. These misses will have a prefetching effect that will speed up the re-execution of *T2*.

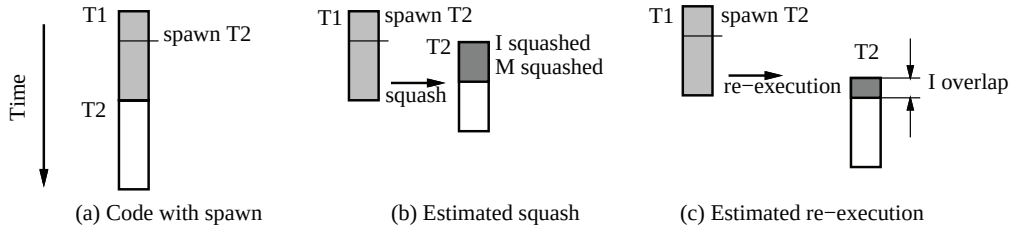

Figure 5.4: Estimating the benefits of a task squash.

The energy profiler deselects tasks that degrade the $E \times D^2$ product. If it estimates that allowing the spawn of *T2* degrades the $E \times D^2$ product resulting from that task, it requests the removal of the spawn.

All tasks have an estimation of the average number of instructions squashed ($I_{squashed}$), average task size ($I_{taskSize}$), the average instruction overlap after re-execution ($I_{overlap}$), and the estimated number of L2 misses ($N_{L2misses}$). The profiler also has an average task spawn overhead ($I_{overhead}$), and the average stall per L2 miss (I_{L2} [1]). Using the previous parameters, for each task, the profiler approximates delay and energy as follows:

$$D = \frac{I_{taskSize} + I_{overhead} - I_{overlap} - N_{L2misses} \times I_{L2}}{I_{taskSize}}$$

[1]The profiler assumes that each instruction takes one cycle.

$$E = I_{taskSize} + I_{squashed}$$

If D is smaller than zero, it is set equal to zero. The profiler removes a task when the resulting $E \times D^2$ product is bigger than $I_{taskSize}$.

5.3.2 Storage and Logic for Data Versioning in the Memory Hierarchy

In SM, there are operations that require changing the tag state of *groups* of cache lines. For example, when a task is squashed, its dirty cache lines are invalidated. Also, in eager-commit systems, when a task commits, its dirty cache lines are merged with main memory through write backs [19] or ownership requests [32]. Finally, in lazy-commit systems, when a cache has no free LIDs (Section 2) left, it needs to recycle one. This is typically done by selecting a long-committed task and writing back all its cache lines to memory. Then, that task's LID becomes free and can be re-assigned.

Proposed SM schemes typically support these group operations with energy-intensive actions or expensive hardware. We want to avoid both.

For some operations, some schemes use a hardware finite state machine (FSM) that, in the background, repeatedly walks the tags of the cache. For example, to recycle LIDs in [28], a FSM regularly selects the LID of a committed task from the LID Table, walks the cache tags writing back to memory the lines of that task, and finally frees up the LID. The FSM operates in the background *eagerly*, using free cache cycles. Other schemes perform similar hardware walks tags while stalling the processor to avoid causing races. For example, to commit a task in [32], a special hardware module sequentially requests ownership for a group of cache lines whose addresses are stored in a buffer. Finally, some schemes use "one-shot" hardware signals that change the tag state of a group of lines in a handful of cycles. For example, [10, 16, 32] do so to invalidate the dirty lines of a squashed task. However, in multi-version caches, this operation may adversely affect the cycle time.

To save energy in these group operations, we propose two energy-centric optimizations.

1. Avoid Eagerly "Walking" the Cache Tags. To save energy, we want to perform all these group operations lazily in the background, especially avoiding any eager walk of the cache tags. Eager operation, even when there are free cycles, consumes energy that may not be fully justifiable.

We only activate an eager background FSM in one case: to recycle LIDs when the cache is about to run out of them. Specifically, we do it when there is only one free LID left. Overall, while this optimization may sometimes improve performance slightly, its main attraction is that it saves energy.

Our optimization relies on the table that contains translations from LIDs to version (LID Table from Section 2). Each entry in the LID Table is extended with summary use information for that LID: the number of lines that the corresponding task has in the cache, and whether the task is killed or committed. With this information, all the operations discussed above are performed in the background, lazily, and without address walking.

For example, consider a task kill or commit. When a task is killed or committed, its LID Table entry is updated by setting the Killed or the Committed bit, respectively (Figure 5.5-(a)). No tag walking is performed. Assume that, later, space is needed in a cache set that has no invalid line. As part of the (off-critical path) replacement algorithm, the LID Table is accessed for the lines in the cache set (Figure 5.5-(b)). For the entries that have the Killed bit set, the count of cached lines is decremented, and the corresponding lines in the cache are either chosen as the replacement victim or invalidated. Also, for the entries with the Committed bit set, the count is decremented, and the lines in the cache are written back to L2 to make room. If any one of these counters reaches zero, that LID is recycled. This enables continuous LID recycling without the need for tag walking.

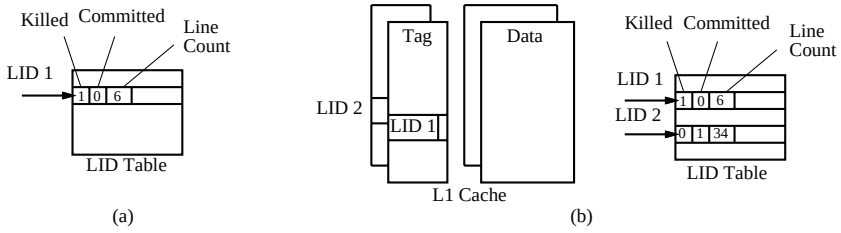

Figure 5.5: Using the LID Table on a task kill (a) and a cache line replacement (b).

2. Low-Energy Reuse of Cached Data on Task Restart. When a task is restarted after a violation, it should be able to reuse any clean lines remaining in the cache from its previous execution with minimal energy. This is relatively easy to do in existing schemes that do not restart a task until the hardware invalidates all the dirty cache lines of the task [10, 32]. Such schemes can simply give the same LID to restarted task, which will then trivially reuse the cached data.

Since our scheme restarts a task immediately, and only invalidates the task's dirty cache lines lazily, we cannot give the same ID to the task — the task would reuse invalid data. However, if we give the task a different LID, it is harder to reuse cached lines. Specifically, on a cache miss, the task needs to access the LID Table to see if any of the clean lines in the target set belong to a previous execution of the same task. Such lines will have a different LID but the same version in the LID Table.

We propose an energy-centric optimization that sometimes eliminates this access to the LID Table. The optimization consists in assigning to each task a combination of LID and *LID Offset (LIDOff)*. A task begins with a LIDOff set to zero; if it gets restarted, it keeps the same LID and increments its LIDOff. Moreover, cache tags include both LID and LIDOff. With this support, if an access finds that a cache line has the same LID and a lower LIDOff, it is a line from a previous execution of the same task. If the line is clean, the access is treated as a hit, and the line is *Promoted* by updating its LIDOff to the current value. The LID Table is not accessed. Avoiding the LID Table access affects performance little, as an out-of-order processor could hide the needed cycle(s). However, it saves energy.

5.3.3 Additional Instructions

Our profiler also prunes tasks that are predicted to be dependence free but are smaller than a certain threshold size. Performance considerations alone would suggest a lower threshold, given that typically, there are free CPUs. However, not spawning them reduces task boundaries and, therefore, code bloat.

5.4 Novel Timestamp Intervals for Task Order Management

In any SM system, tasks have a relative order, which they explicitly or implicitly embed in the CMP protocol messages they issue and the cached data they own. Such order is most obviously needed when two tasks communicate. For example, consider a task reading cached data produced by a second task. The relative order of the tasks is assessed, and the datum is provided only if the former task is a successor of the latter. Analogously, contemplate an invalidation message from a task to data read by a second task. The task order is considered and, if the reader is a successor,

a dependence violation is triggered.

Under in-order task spawn, Version ID assignment is easy: since tasks are created in order, it suffices to assign monotonically increasing timestamps or versions to newer tasks. A parent gives to its child its timestamp plus one. With this support, tasks with higher timestamps are successors of those with lower ones.

Unfortunately, such an approach does not work when tasks are created out of order. To maintain order now, we propose to represent a task with a *Timestamp Interval*, given by a *Base* and a *Limit* timestamp ({B,L}). Both base and limit timestamps are operated upon in a task spawn. Specifically, when a task spawns a child, it splits its Timestamp Interval in two pieces: the higher-range subinterval is given to the child (since it is more speculative), while the lower-range subinterval is kept by the parent. With this support, protocol messages and cached data are directly (or indirectly) associated with the base timestamp. When communication between tasks occurs, the base timestamps of the two tasks are compared *exactly* as in the in-order case.

As an example, Figure 5.6-(a) shows a program with a call to subroutine *S1*, which in turn calls *S2*. Assume that we use three tasks: task i executes the non-speculative code, j executes the continuation of *S1*, and k executes the continuation of *S2*. The resulting task tree is shown in Figure 5.6-(b), while Figure 5.6-(c) shows the Timestamp Intervals of each task.

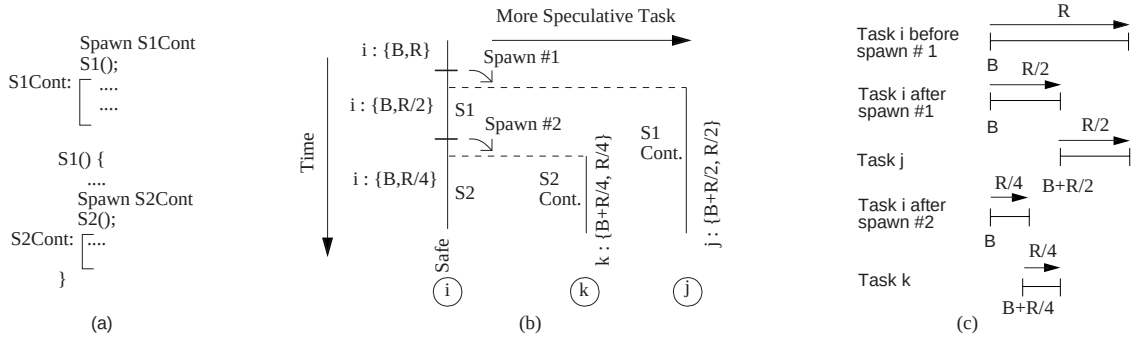

Figure 5.6: Changes in the base and range timestamps when tasks are spawned.

The example assumes that the initial interval for task i is {B,L}, and that intervals are partitioned in half. When i spawns j, i keeps $\{B,\frac{L}{2}\}$ and j obtains $\{B+\frac{L}{2},\frac{L}{2}\}$. When i later spawns k, i retains $\{B,\frac{L}{4}\}$ and k obtains $\{B+\frac{L}{4},\frac{L}{4}\}$. With this scheme, as we move from safe to most speculative task following sequential order (i, k, and j), we encounter adjacent intervals ($\{B,\frac{L}{4}\}$, $\{B+\frac{L}{4},\frac{L}{4}\}$,

$\{B+\frac{L}{2},\frac{L}{2}\}$) with increasing base timestamps.

In general, a simple approach is to give $\frac{1}{2}$ of the current interval to the child. Nevertheless, since a task does not spawn many children, we experimentally found that it is better to keep $\frac{1}{4}$ of the interval, and give $\frac{3}{4}$ to the spawn child. Additionally, there is one case where we can be more efficient. If the parent predicts that this is its own last child, the parent keeps $\frac{1}{64}$ instead of keeping $\frac{1}{4}$. We propose to predict these cases with a small per-processor Interval Distribution Predictor (IDP). The IDP is indexed by the start PC of the task. An entry contains the number of children spawned by the task when it last ran. The IDP is updated at task end and is read at task spawn. With the IDP, we can catch the common case of tasks that spawn a single child. This case occurs often in tasks from loop iterations. If the IDP misses, we simply give $\frac{1}{4}$ of the interval to the child. The IDP design is shown in Figure 5.7. It is a small cache-like structure with a 2 saturating counter (nSpawns in Figure 5.7). Note that some of these efficiencies could also be obtained by using static information gathered by the compiler.

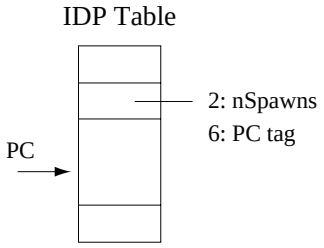

Figure 5.7: Interval Distribution Prediction Table. nSpawns is a saturated counter for the number of spawns performed by a given task.

The evaluated and proposed SM architecture does not support selective restart because we observed that the additional complexity was not worth the increase in performance. Nevertheless, if the out-of-order spawn framework were to support selective restart, the proposed Timestamp Intervals would work with minor modifications. When a task is killed, its timestamp interval could be consolidated in its killer's. Since a task can only be killed by its immediate predecessor in sequential order (Section 5.5), the combination of both intervals results in another contiguous range of timestamps. Specifically, the killing task keeps its old B and sets L to the sum of both tasks' L. For the example in Figure 5.6, if task j is killed, k becomes $\{B+\frac{L}{4},\frac{3L}{4}\}$; if j and k are killed, i becomes {B,L}.

Our scheme assigns no L to the most speculative task, as it implicitly takes the maximum value (L_{max}). This allows the system to dynamically expand the range of used timestamps. Indeed, when the most speculative task spawns a child, it keeps the range {B,L_{max}} for itself, and sets the base of the child to B+L_{max}. The child is now the new most speculative task.

Note that, it is possible that a program causes the timestamps to wrap around. In addition, in rare cases, a task may reach a point where it needs to spawn a child and its interval has size 1. These cases are discussed in Section 5.4.1.

Our scheme has some resemblance to Cleary *et al.*'s virtual sequences [12]. The latter have significant implementation limitations (Section 9).

5.4.1 Special Cases in Timestamp Intervals

There are two infrequent, special cases when handling Timestamp Intervals. The first one is when a task wants to spawn a child and has no interval to assign. In this case, it simply sends a kill to all the successors, making the task the most speculative one. At this point, the task can obtain as many timestamps as needed (Section 5.4).

The second case is when a program exhausts the physically representable timestamp range. Our solution is to recycle old timestamps in chunks. For that, we divide the whole representable timestamp range into four chunks, based on the two most significant bits of B. When all the tasks with intervals in the lowest chunk (e.g. the 00 chunk) have committed, we recycle the chunk. This involves sending a reprogramming signal to the logic of the timestamp comparators so that timestamps in the recycled chunk are now the highest (i.e. 00 is more speculative than 11). Then, we can start assigning timestamps from the chunk to newer tasks.

The reprogramming signal is issued in the infrequent case that a task with an interval that straddles two chunks commits. With this approach, all the tasks in the CMP can at most use $\frac{3}{4}$ of the whole timestamp range at a time. To see how many tasks can be concurrently supported, assume that B and L have b and l bits, respectively. If, in the worst case, each task has a single child, and the child is given the maximum timestamp range possible (2^l), the maximum number of tasks is then $\frac{3}{4} \times 2^{b-l}$. Consequently, if we want to support about 20 concurrent tasks, $b-l$ should be at least 5.

5.5 Immediate Successor List for Task Squash and Commit

In SM, a task must be able to find its immediate successor very quickly, to perform the time-critical operations of commit and squash. Specifically, when the safe task commits, it passes the commit token to its immediate successor, which may be waiting for it to commit. For squash, a task is squashed when it reads data prematurely (data violation) or is spawned in the wrong branch path (control violation). For a data violation, the victim task receives a restart signal, which induces the destruction of the state modifications and restarts task execution from its beginning – hoping that the re-execution will read correct data. In either case, a kill signal is also sent to the immediate successor of the victim task and, recursively, to the immediate successor of that one up until the most speculative task. This ensures that all possible side effects of the victim task are erased.

Under in-order task spawn, it is easy to find a task's immediate successor and, recursively, immediate successors until the most speculative task. For example, consecutively spawned tasks are often allocated on contiguous processors, making it trivial to identify the immediate successor. In other designs, a table with immediate successor information is used, which is easy to maintain because only one task can spawn at a time. Finally, any scheme used is likely to be largely free of protocol races, as only one task spawns at a time.

Under out-of-order task spawn, identifying the immediate successor and all the more speculative tasks is not straightforward. For example, in Figure 3.1-(e), if task *7* is killed, it is not trivial for it to identify and kill tasks *4*, *3*, *5*, and *2*, which were created before and independently of *7*. Moreover, any solution has to be carefully crafted to avoid inducing races in the SM protocol of the distributed CMP if multiple operations happen concurrently. Finally, since commit and squash are time-critical, we cannot use a solution based on repeated comparison of timestamps.

To support efficient and race-free commit and squash, we propose that the tasks dynamically link themselves in hardware in a list according to their sequential order. We call this list the *Immediate Successor* (IS) list. To build the IS list, we add a hardware pointer called the IS pointer to each task structure. We leverage the fact that, at the time of the spawn, (i) the child is always the immediate successor of its parent, and (ii) the child inherits the parent's immediate successor. Consequently, on a spawn, the child receives the parent's IS pointer, and the parent sets its IS pointer to point to the child. In the example of Figure 3.1-(e), the IS list links *1* to *6*, *6* to *7*, *7* to

4, and so on.

When a task kills all its successors, its IS is set to nil. Consequently, Task *2*'s IS pointer in Figure 3.1-(e) is nil.

With this support, when a task needs to pass the commit token, it uses the IS list. Moreover, when a squashed task needs to kill all its successors, it sends a kill signal with its own identity downstream the IS list. All successors are killed in turn. When the kill signal reaches a task with a nil IS, an acknowledgment is sent to the originating task, which sets its IS to nil. The result is very fast commit and squash. In addition, the SM protocol implementation is simplified in a major way: even when multiple kill signals occur concurrently, since all signals are serialized along the same path, protocol races are minimized.

5.6 Dynamic Task Merging for Efficient Resource Allocation

In SM systems, tasks compete for CMP resources such as CPUs, on-chip contexts, and cache space. Under out-of-order task spawn, such competition is harder to manage than under in-order spawn. The reason is that highly-speculative tasks may hog resources and starve more critical (less speculative or even safe) tasks that are spawned later. For example, in Figure 3.1-(e), when safe task *1* is about to spawn *6*, all the CPUs and contexts in the CMP may be in use by more speculative tasks *4*, *3*, *5*, and *2*.

To allocate chip resources efficiently, we propose a new CMP microarchitectural technique that we call *Dynamic Task Merging*. It consists of transparent, hardware-driven merging of two consecutive tasks at run time. The merging may occur before or after the second task has been spawned. In effect, it enables the machine to prune some branches of the task tree based on dynamic load conditions. The overall effects of dynamic task merging are an increase in the size of the running tasks and a reduction in their dynamic number.

These effects increase execution efficiency in several ways. First, highly-speculative tasks can be merged, therefore freeing resources for more critical tasks. Second, with large tasks, the overhead related to task spawn has a relatively lower weight, and both caches and branch predictors work better, as a CPU reuses their state for a longer time. Finally, given that the hardware can adjust the number of tasks at run time, the SM compiler can be more aggressive at creating tasks, which

may ultimately lead to higher performance.

Given a pair of tasks, we propose two types of dynamic task merging, depending on whether or not the second task has been spawned. If it has not, dynamic task merging typically involves skipping the spawn instruction of the second task and the task-end instruction of the first task. If the second task has already been spawned, dynamic task merging typically involves killing it and skipping the task-end instruction of the first task.

The first type of task merging can be triggered on any task when it is about to spawn a child. We call it *MergeNext*. The second type of task merging can be triggered on any pair of consecutive tasks in the CMP at any time. However, to maximize efficiency and simplify the implementation, we only trigger it on the two most speculative tasks in the CMP. Consequently, we call it *MergeLast*. Usually, we do it when a new task is about to be spawned somewhere in the CMP.

Note that *MergeNext* and *MergeLast* are not exclusive choices. Overall, every time that a task finds a spawn instruction, we select one of four possible choices: spawn normally, MergeNext, spawn and MergeLast, and both MergeNext and MergeLast. Figure 5.8 shows the choices when task *4* finds the spawn for *5*. In the rest of this section, we discuss the microarchitecture support for MergeNext and MergeLast, and the heuristics that we use to decide which of the four choices to select. Some compiler implementation details are discussed in Section 6.

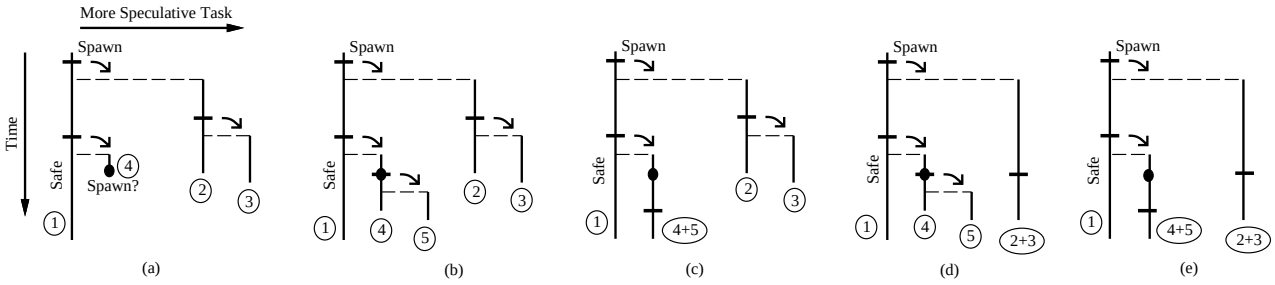

Figure 5.8: Choices when task *4* finds the spawn for *5*: spawn (b), MergeNext (c), spawn and MergeLast (d), and MergeNext and MergeLast (e).

5.6.1 MergeNext Microarchitecture

A task initiates a MergeNext by skipping a spawn instruction. After that, in the simplest case, the task will also have to skip the first task-end instruction that it finds, and finish only when it finds the second task-end. In general, if a task initiates N MergeNext operations by skipping N spawns,

it will have to also skip *N* task-ends and complete only when it finds the *N+1* one.

Consider now a task that skipped a spawn in a MergeNext and later spawns a child. In this case, since the child is more speculative, the responsibility to complete the task merge is "passed on" to the child: the child will skip the first task-end that it finds and finish only at the second one. As for the parent, it simply finishes at the first task-end that it finds.

The microarchitecture needed to support MergeNext is a counter in the processor called *Number of Ends to Skip* (NES). The NES belongs to the running task, and is checked and modified in hardware. The NES pointer is stored in the TaskHolder. Specifically, when a task initiates a MergeNext, the NES is incremented. When a task finds a task-end instruction, the NES is checked. If it is non-zero, it is decremented and the end instruction is skipped. Otherwise, the end is executed. Moreover, when a task spawns a child, its NES is copied to the child's and is then cleared. The child now owns the merges.

A task's NES is affected by two more events. First, when a task becomes the most speculative one (its IS pointer becomes nil), its NES ceases to matter — the task simply skips any task-end instruction that it finds. This is the appropriate behavior for the most speculative task, which should not be stopped by end instructions. However, if the task spawns a child, the NES of both tasks are updated as usual. The second special event occurs when a task gets restarted (Section 5.5). In this case as the task recovers its initial state, it also recovers its initial NES.

5.6.2 MergeLast Microarchitecture

MergeLast involves killing the most speculative task in the CMP and ensuring that, when the new most speculative task completes its own code, it executes the code of the killed task.

The microarchitecture needed to support MergeLast is the IS list (Section 5.5). A task initiates a MergeLast by sending a MergeLast hardware signal down the IS list. Each task in the list passes, in hardware, the signal and its own identity to its successor. When the signal reaches a task with a nil IS pointer, that task sends an acknowledgment to its immediate predecessor (whose identity it knows) and terminates. The immediate predecessor sets its IS pointer to nil, as it is now the most speculative task. No other action is necessary. When the latter task reaches its end, it will skip it and continue executing, effectively merging its code with that of the killed task. This is because,

as discussed in Section 5.6.1, a task with a nil IS pointer skips task-ends.

Note that the operation of a task killing all its successors after a violation (Section 5.5) is similar to a MergeLast except that all the tasks downstream the IS list are killed. In fact, to keep the hardware simple, we implement such an operation as a set of MergeLast operations: the killing task keeps issuing MergeLast operations until it becomes the most speculative task.

5.6.3 Task Merge Heuristics

Every time that a task finds a spawn instruction, decisions on task merging are made. To keep the hardware simple and the overheads low, this thesis proposes a simple decision algorithm.

The algorithm is based on two notions. First, we conservatively assume that any running task, even if highly speculative, is likely to perform useful work. Consequently, we try to avoid killing tasks. Second, we rely on squash information to reduce useless work. Specifically, if a task has been restarted twice due to violations, it is not allowed to get a CPU anymore. It simply remains in one of the several on-chip task contexts until it becomes safe. This policy prevents highly-speculative, frequently-squashed tasks from clogging the CPUs. It also allows the hardware to estimate the level of load in the CMP by examining the fraction of on-chip task contexts that are in use.

With this support, we use the following algorithm. We use the CPU usage to decide on MergeNext. If all CPUs are busy, since they appear to do useful work, we perform MergeNext. However, every $Th_{MaxMNext}$ MergeNexts, we skip one to prevent tasks from becoming so large that a squash would be very costly.

As for MergeLast, we decide based on the estimated use of on-chip task contexts. If most of them are used, it is likely that many highly-speculative, frequently-squashed tasks are waiting. In this case, one could be killed with little performance penalty. While we could perform a MergeLast only when no context is free, the operation would then be in the critical path. Consequently, we use a threshold: if the estimated number of used contexts is over Th_{MLast} at the time of a spawn, we perform MergeLast.

Chapter 6

Compilation Support

For this thesis, a new fully automated SM compiler has been built. It generates in-order and out-of-order tasking out of sequential, integer applications. The compiler adds several passes to gcc 3.5. This version of gcc uses a static single assignment tree as the high-level intermediate representation [25]. Building on this software allows us to leverage a complete compiler infrastructure. For example, we annotate the control flow graph structure with high-level information as we generate the tasks. Also, working at this high level is better than using a low-level representation such as RTL: we have better information and it is easier to perform pointer and dataflow analysis. At the same time, our transformations are much less likely to be affected by unwanted compiler optimizations than if we were working at the source-code level.

The resulting code quality when SM is disabled is comparable to the MIPSPro SGI compiler for integer codes at the O3 optimization level. This is because, in addition to using a much improved gcc version, we also use SGI's source-to-source optimizer (copt from MIPSPro). The latter performs PRE, loop unrolling, inlining, and other optimizations. When SM is enabled, code quality is necessarily lower due to the code being partitioned into tasks.

6.1 Compiler Phases

Each SM task corresponds to a subset of the execution of the program. A task is called from a spawn point; this is the point in execution that starts the task. A task starts executing at a begin point, which is the first instruction of the executing task. A task has only one begin point and only one spawn-point. However, multiple ending points are possible. At runtime, an instance of a

(static) task, defined by some begin point, will have a single end point, but each (dynamic) instance of the task may take a different path through the code, sharing only the sequential code following the begin point.

A SM compiler consist of four main phases: Task selection, spawn hoisting, task pruning, and live-in generation. Once a task is selected, the spawn point is hoisted as much as possible. Task with little potential are eliminated by the task pruning pass. The live-ins are calculated for the remaining tasks. [39] has a more detailed information about the algorithms used.

6.1.1 Task Selection

Task selection for SM compilers is easier than in automatic parallelizing compilers. Because dependences are allowed to exist between tasks, a variety of heuristics can be used to choose tasks. Ideally, these heuristics capture some basic properties that tasks should have: few intertask dependences, enough work to overcome overheads, and few live-ins. Choosing tasks which meet these criteria and provide the optimal performance improvement is NP-hard [4].

Our compiler uses the following modules as potential tasks for both the in-order and out-of-order environments: subroutines from any nesting level, their continuations, and loop iterations from multiple loops in a nest. All subroutines are potentially chosen unless they are very small. Recursion is handled seamlessly. In loop nests, the compiler makes decisions based on loop iteration size, which has to be larger than a certain minimum.

The actual tasks that make it to the final binary are different in the in-order and out-of-order environments. The out-of-order pass can select all the tasks mentioned, subject to some pruning heuristics, without worrying about the number of children per task. The in-order pass has to be more careful, since a task can only have a single child. Consequently, the in-order pass has an initial step where it analyzes all the files in the program and generates a complete task call graph. Then, using heuristics about task size and overheads, it eliminates tasks from the graph until each task only has a single child. We trust the quality of our heuristics based on the fact that the resulting in-order SM code obtains speedups comparable to previous work [44].

As an example, Figure 6.1 shows how the compiler generates out-of-order tasks out of a subroutine and its continuation. Chart (a) shows the dynamic execution in and out of the subroutine.

The compiler marks the subroutine and continuation as tasks, and inserts two spawn instructions in the caller (Chart (b)).

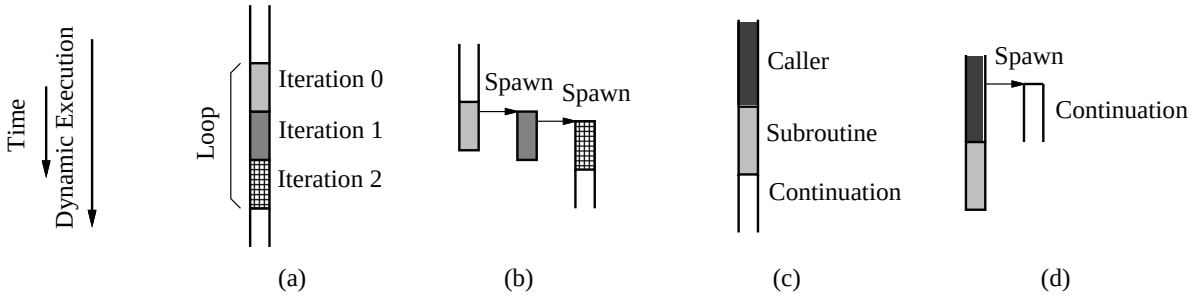

Figure 6.1: Generating tasks out of a subroutine and its continuation.

6.1.2 Spawn Hoisting

Task selection defines where tasks begin and end execution. Spawn hoisting selects the place where the spawn instruction is placed. The spawn point dictates when a task can begin executing. Spawn hoisting is the act of moving the spawn point earlier in the code to create parallelism.

A spawn is hoisted as far up in the source code as possible, as long as the new position is execution equivalent with the start of the task to spawn. We do not hoist above statements that can cause data or control dependence violations. Under out-of-order spawn, we make sure that the tasks are spawned in reverse order. Under in-order spawn, a spawn cannot be hoisted above the caller task.

Looking at Figure 6.1, we can see that the compiler hoists the spawn for the continuation (Chart (c)) and subroutine (Chart (d)). As usual, tasks on the right side are more speculative.

6.1.3 Task Pruning

Task selection selects all the possible tasks. The task pruning pass of the SM compiler deselects tasks. It tries to maximize performance and minimize energy waste by eliminating tasks that offer no potential or high cost.

Parallelism

A necessary condition for parallelism is starting tasks well before they would be executed sequentially. By allowing them to execute earlier, parallelism with older tasks is enabled. Without a large distance between the spawn point and the begin point, there can be no speedup due to parallelism. Therefore, tasks with little hoisting are eliminated.

Small Tasks

Eliminating small tasks is an important technique to guarantee good code quality. Creating tasks too frequently degrades code quality. The only exception are small tasks with high L2 miss rate.

Few Live-ins

The number of registers that need to be passed to a task significantly affects performance. Some research has gone to great lengths to make registers available as soon as possible [32], or propose algorithms that minimize the live-ins passed [18]. For the task boundaries selected, we found that task size are in the hundreds of instructions with few live-ins.

Few Intertask Dependences

Minimizing memory dependences between tasks further complicates deciding task boundaries. Memory dependences are harder to handle at compile time because tracking them correctly is dependent on the quality of alias analysis in the compiler. Algorithms have been proposed that use a probabilistic measure of the likelihood of dependences between threads since the hardware will enforce any that are missed.

For simplicity reasons, the SM compiler guesses that memory references do not produce data dependence violations. Future enhancements with probabilistic pointer analyses [9] would further improve tasks selection.

6.1.4 Live-in Generation

For the purposes of this compiler, live-ins come from two sources at execution time: registers and memory. Together, the register file and memory describe the complete state of an application.

The SM compiler does not need to worry about memory live-ins because the hardware enforces the dependences. The compiler only needs to consider register file live-ins. Conceptually, at task boundaries all the registers must be spilled.

One of the main performance overheads in the SM compiler comes from converting register dependences into memory live-ins. The most obvious implication is that temporary values must now be pushed and popped from the stack if they are needed in multiple tasks. This is especially costly in loops that incur this overhead for each loop iteration.

Live-ins can never be fully eliminated in a SM environment. However, their impact on performance may be mitigated through several techniques. Value prediction provides a guess of a probable value of a dependence (Section 6.2). If the value can be guessed in advance, the program can avoid waiting for the dependence to be resolved. Another technique is to maximize the distance between producers and consumers of a value. This gives a greater likelihood that the consuming task will receive the correct value. Finally, a simpler technique is to avoid unnecessary loads and stores by eliminating redundant pushes and pops to the stack.

Task Merging Support

With task merging, as a task completes its code, it goes on executing the code of its immediate successor. This means that the task must have a way of obtaining the live-in register values for its continuation code. With our compiler, this is possible: all register values changed by a task that may be used by successors are stored in memory when the task finishes. Moreover, all the live-ins of a task are read from memory. Consequently, as a task merges with its successor, it automatically reads from memory the live-ins of the successor.

6.2 Value Prediction

A general approach for deciding when value prediction should be used is difficult for a compiler, but there are some characteristic locations that have been considered for value prediction in previous studies [26, 21, 43]. Function return values have long been considered an ideal place to predict values since functions tend to only contribute an additional live-in to the function continuation. Also, loop induction variables are highly predictable and tend to be carried across loop-task boundaries.

To support value prediction, we devise a simple scheme that leverages SM hardware. It uses a global non-versioned (plain) shared memory location to hold the predicted value (*prediction*) and architectural support for silent stores. Silent stores have been proposed to reduce dependence violations in SM [33]. If silent stores are unavailable, our scheme still works but has higher overhead.

6.2.1 Return Value Prediction

The variables returned by functions are typically read too early by hoisted continuation tasks. As a result, when the function finally updates the variable, the continuation gets squashed. However, we can eliminate many violations by predicting that the value returned by the function will be the same as the last time it was called.

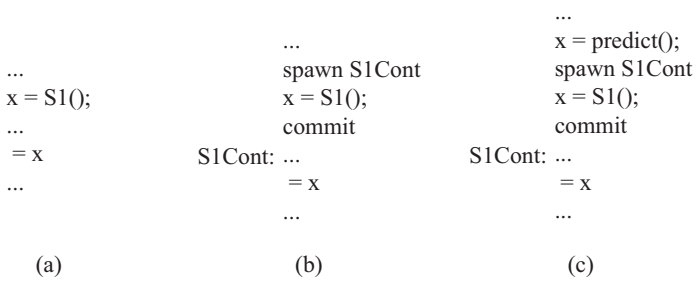

Figure 6.2: Code generated for a subroutine and its continuation. (a) Original code, (b) instrumentation without value prediction, (c) instrumentation with value prediction.

Figure 6.2 shows how return value prediction is done in our compiler. Figure 6.2-(a) shows the original subroutine. Figure 6.2-(b) shows the code generated by the SM compiler without value prediction. Figure 6.2-(c) shows the code generated with value prediction. Before the continuation is spawned, the variable that will receive the function's result (x) is set to the predicted value. Variable x can then be read by the continuation. When the function finally returns and sets x, two things may occur. If the value is the predicted one, the store is silent and induces no squash; if the value is different, it causes a cache line invalidation that automatically squashes the continuation. The code also updates the global *prediction* memory location with the return value. As mentioned, this update causes no squashes of any other tasks because we use a non-versioned variable. Alternatively, we could use an entry in a distributed hardware prediction table. For best results in our experiments, we have a *prediction* variable for each function call site.

If the machine does not support silent stores, we store the return value of the function into a

scratch variable. Then, we check in software if the scratch variable contains the predicted value. If not, the scratch variable is written to the *prediction* and x variables, triggering a squash.

6.2.2 Induction Value Prediction

Most loops have induction variables. All induction variables are task live-ins. Without value prediction, all the loop tasks have a guaranteed restart. To avoid this case, the compiler breaks the induction variables by value predicting the increment. The scheme is similar to the return value prediction, the major difference is that instead of predicting the induction variable value, the increment is predicted.

6.3 Profiler

The compilation process includes a simple profiler. The profiler takes the initial SM executable and identifies those tasks that should be eliminated because they are likely to induce harmful squashes according to our models. The profiler returns the list of such spawns to the compiler. Then, the compiler generates the final executable by removing those spawns and integrating the target tasks of those spawns with their statically predecessor tasks. On average, the profiler takes few minutes to run on a 3GHz Pentium IV machine.

The profiler executes the binaries sequentially, using the Train data set for SpecInt codes. As the profiler executes a task, it records the variables written. As it executes tasks that would be spawned earlier, it compares the addresses read against those written by predecessor tasks. With this, it can detect potential violations. The profiler also models a cache to estimate the number of misses in the real machine's L2, although no timing is modeled.

6.3.1 Performance Profiler

The profiler identifies those spawns where the ratio of squashes per task commit is higher than R_{squash}. For those spawns, it estimates $I_{squashed}$, $I_{overlap}$, and $M_{squashed}$ as in Figure 5.4. By doing so, it estimates the performance benefit that a task squash brings. Some benefit comes from the data prefetching provided by cache misses recorded before the task is squashed ($M_{squashed}$). Other

benefit comes from true overlap of the instructions in the task with other tasks, as the task is re-executed after the squash ($I_{overlap}$). With these measurements, the profiler requests spawn removal if $T_I \times I_{overlap} + T_0 \times M_{squashed}$ is less than a threshold T_{perf}. In the formula, T_0 is the estimated stall per L2 miss, and T_I is the estimated execution time per instruction.

6.3.2 Energy Profiler

The performance profiler identifies those spawns that have a squash per commit rate higher than R_{squash}. When the energy profiler is activated a more conservative value is selected for R_{squash}.

The energy profiler removes tasks if subtracting $T_I \times I_{overlap} + T_0 \times M_{squashed}$ from the program time and adding $I_{squashed} \times E_0$ to the program energy, increases the program's $E \times D^2$ product. The values for the thresholds and parameters used are listed in Section 7.

Chapter 7

Evaluation Methodology

To evaluate the energy and performance of SM, we compare a SM chip with multiple narrow-issue processors to a non-SM chip with a single conventional, wide-issue processor.

A new architectural simulator has also been developed for this thesis. The simulator uses MIPS ISA execution-driven simulations with detailed models of out-of-order superscalar processors and memories, enhanced with models of dynamic and leakage energy from Wattch [5], Orion [41], CACTI [30], and HotLeakage [45].

7.1 Architectures Evaluated

The SM CMP proposed has four 3-issue cores as explained in the microarchitecture of Section 5. We call the chip *SM4-3i*. The non-SM chips have a single superscalar with a conventional L1 and L2 on-chip cache hierarchy. We consider two: one is a 6-issue superscalar (*Uni-6i*) and the other a 3-issue superscalar (*Uni-3i*). We choose to compare the *SM4-3i* and *Uni-6i* designs because both chips have approximately the same area, as can be estimated from [20, 30].

Table 7.1 shows the parameters for *SM4-3i* and *Uni-6i*. As we move from 3-issue to 6-issue cores, we scale all the processor structures (e.g., ports, FUs, etc) according to the issue width of the core. We try to create a balanced processor by scaling up the processor resources while minimizing $E \times D^2$ as much as possible.

In our comparison, we favor *Uni-6i*. We assume that *Uni-6i* has the same frequency and the same pipeline depth as the cores in *SM4-3i*. This helps *Uni-6i* because, in practice, a 6-issue core would not cycle as fast as a 3-issue core with the same pipeline. For example, according to

Processor Parameters	PROPOSED: *SM4-3i*	COMPETITION: *Uni-6i*
Cores/chip	4	1
Running tasks/core	1	1
SM hardware?	Yes	No
Frequency, technology	**5 GHz, 70 nm**	**5 GHz, 70 nm**
Fetch, issue, retire width	6, **3**, 3	6, **6**, 6
ROB, I-window size	126, 68	204, 104
LD, ST queue	48, 42	66, 54
Mem, int, fp units	1, 2, 1	2, 5, 2
Branch predictor:		
Penalty	14 cycles	14 cycles
BTB	2 K, 2 way	2 K, 2 way
global gshare(11)	32 Kbits	32 Kbits
local 2 bit	32 Kbits	32 Kbits
L1 cache:		
size, assoc, line	16 KB, 4, 64 B	16 KB, 4, 64 B
OC, RT	1, **3**	1, **2**
RT remote L1 (min)	8 cycles	—

Tasking Parameters
Task containers/processor: 8
B, R timestamp size: 32, 22 bits
NumMNext : 8
Latencies in cycles (min):
From spawn to new thread: 14
From violation to full kill notification: 20
Drain proc pipeline: 14
Fraction of interval given to child: 3/4
R_{squash}: 0.8 ; T_0: 200 cyc; T_{perf}: 100 cyc
E_0: 8pJ; S_{perf}: 46; S_{ener}: 54

Common Memory System
I-L1 cache size, assoc, line:: 16KB, 2, 64B
OC, RT: 1, 2
L2 cache size, assoc, line: 1 MB, 8, 64 B
OC, RT: 1, 11
Mem bandwidth, RT: 10 GB/s, 500 cycles

Table 7.1: Architectures considered. In the table, OC and RT stand for occupancy and minimum-latency round trip from the processor, respectively. All cycle counts are in processor cycles. In our comparison, we use the same processor frequency for both *Uni-6i* and *SM4-3i*.

CACTI [30], the access time of the register file in *Uni-6i* would be nearly double that of the *SM4-3i* cores. In our simulations, we assume the same frequency for *Uni-6i* and *SM4-3i*.

Since both processors have the same pipeline depth and branch misprediction penalty, we feel that it is fair to also give them the same branch predictor. In addition, both processors have an integer and a FP FU cluster. Since we run integer codes in the evaluation, the FP cluster is clock-gated almost all the time.

The tag array in *SM4-3i*'s L1 caches is extended with the LID, and the Write and Exposed-Read bits (Figure 5.3). At worst, the presence of these bits increases the access time of the L1 only slightly. To see why, note that the LID bits can simply be considered part of the line address tag, as a hit requires address and LID match. Moreover, in our protocol, the Write and Exposed-Read bits are not checked before providing the data to the processor; they may be updated after that. However, to be conservative, we increase the L1 access latency in *SM4-3i* one cycle over *Uni-6i*, to *3 cycles*.

Uni-3i is like *Uni-6i* except that the core is 3-issue, like those in *SM4-3i*, and the L1 cache only has 1 port. For completeness, we also evaluate one additional chip: *SM2-3i*. *SM2-3i* is a SM CMP like *SM4-3i*, but with only two cores.

7.2 Energy Considerations

We estimate and aggregate the dynamic and leakage energy consumed in all chip structures, including processors, cache hierarchies, and on-chip interconnect. For the dynamic energy, we use the Wattch [5] and Orion [41] models. We apply aggressive clock gating to processor structures. In addition, unused cores in the SM CMP are also clock gated. Activating and deactivating core-wide clock gating take 100 cycles each. Clock-gated structures are set to consume 5% of their original dynamic energy, which is one of the options in Wattch. We extend the Wattch models to support our deeper pipelines and to take into account the area when computing the clock energy. The chip area is estimated using data from [20] and CACTI [30].

Leakage energy is estimated with HotLeakage [45], which models both sub-threshold and gated leakage currents. We use an iterative approach suggested by Su *et al.* [35]: the temperature is estimated based on the current total power, the leakage power is estimated based on the current temperature, and the leakage power is added to the total power. This is continued until convergence. The target average temperature at the junction for the worst application is 85 C, as recommended by the SIA Roadmap [1].

From our calculations, the average power consumed by the *Uni-3i* and *Uni-6i* chips for the SpecInt 2000 applications is 32 and 60 W, respectively (more data will be shown later). Of this power, leakage accounts for 35% and 30%, respectively. The majority of the power increase from *Uni-3i* to *Uni-6i* is due to five structures that more than double their dynamic contribution, largely because they double the number of ports. These are the rename table, register file, I-window, L1 data cache, and data TLB. In addition, the data forwarding bus also increases its dynamic contribution by 70%. We base our confidence in the accuracy of these numbers on the fact that Wattch and HotLeakage have been validated for similar superscalars [5, 45]. The additional structures added by the SM CMP are largely regular SRAM structures (Section 5.1). Such structures can be easily modeled by CACTI, Wattch, and HotLeakage.

7.3 Applications Evaluated

Our architectures run SpecInt 2000 applications with the Ref data set. The exceptions are *eon*, *gcc*, *perlbmk* (where the compiler fails), and *vortex* (where the simulator fails). *Uni-3i* and *Uni-6i* always run SpecInt 2000 binaries compiled with our SM pass disabled. The code quality is comparable to the MIPSPro SGI compiler for integer codes at O3 level.

SM2-3i and *SM4-3i* run different types of SM binaries compiled. As shown in Table 7.2, we compare four different SpecInt binaries: unmodified binaries (*BaseApp*), SM with in-order spawning (*InOrder*), SM with out-of-order spawning (*OutOrder*), and SM with out-of-order spawning and energy profiler (*Power*).

Name	SM?	Description of Binary
BaseApp	N	Out-of-the-box, sequential version compiled with *O2*. No SM instrumentation
InOrder	Y	In-order task spawning. Selects the same tasks as *OutOrder*. Uses interprocedural analysis pass to eliminate tasks that violate the in-order spawning requirement.
OutOrder	Y	Our proposed out-of-order task spawning. Spawns to: (1) a procedure call (2) continuation of any procedure, and (3) iterations from multiple loops in nest
Power	Y	Our proposed energy profiler on top of out-of-order task spawning. Spawns to: (1) a procedure call (2) continuation of any procedure, and (3) iterations from multiple loops in nest

Table 7.2: Versions of the SpecInt 2000 binaries executed.

These binaries are very different. Specifically, the SM passes re-arrange the code into tasks and adds extra instructions for spawning and commit. In addition, these transformations obfuscate some conventional compiler optimizations, sometimes rendering them less effective. Consequently, to accurately compare the performance of the different binaries, we cannot simply time a fixed number of instructions. Instead, we insert "simulation markers" in the code, and simulate for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), we execute up to a certain number of markers for all binaries, so that the *BaseApp* binary graduated more than 700 million instructions.

Chapter 8

Evaluation

This thesis proposes energy and performance optimizations. The evaluation starts with performance optimizations, and continues with energy/power optimizations.

The evaluation chapter is organized as follows: Section 8.1 shows the overall speedups when out-of-order spawn is activated; Section 8.2 gives insights about the results achieved; Section 8.3 characterizes the architectural characteristics related to the SM sources of energy consumption; Section 8.4 shows the energy cost of SM; Section 8.5 shows the impact of the energy-centric optimizations proposed; Section 8.6 compares the energy consumed of *SM4-3i* with *Uni-6i*; and finally Section 8.7 provides an overall comparison of performance and power.

8.1 Overall Speedups

To evaluate the proposed SM architectures, *SM2-3i* and *SM4-3i*, we use the three types of SM binaries shown in Table 7.2: *InOrder*, *OutOrder*, and *Power*. To evaluate out-of-order spawn for SM, we compare the execution time of the *InOrder* and *OutOrder* binaries running on the *SM4-3i* and the *SM2-3i* architectures. For comparison purposes, we also measure the execution times of the *BaseApp* binary running on the *Uni-3i* and *Uni-6i* architectures. The comparisons to *Uni-3i* and *Uni-6i* show the speedup of SM relative to a single processor of the same width and a wider one, respectively, always under the same frequency. Finally, we also run *OutOrder* on *SM2-3i*, to assess the effect of the number of processors in the CMP.

Figure 8.1 shows the speedups of the different binary-architecture combinations relative to *BaseApp* running on *Uni-3i*. The figure shows speedups for each application and the *geometric*

mean. On top of some bars, we show the speedups. The dots on some bars will be discussed later. The average IPC of each application for *Uni-3i* and *SM4-3i*$_{OutOrder}$ is shown in Columns 2 and 3 of Table 8.1, respectively.

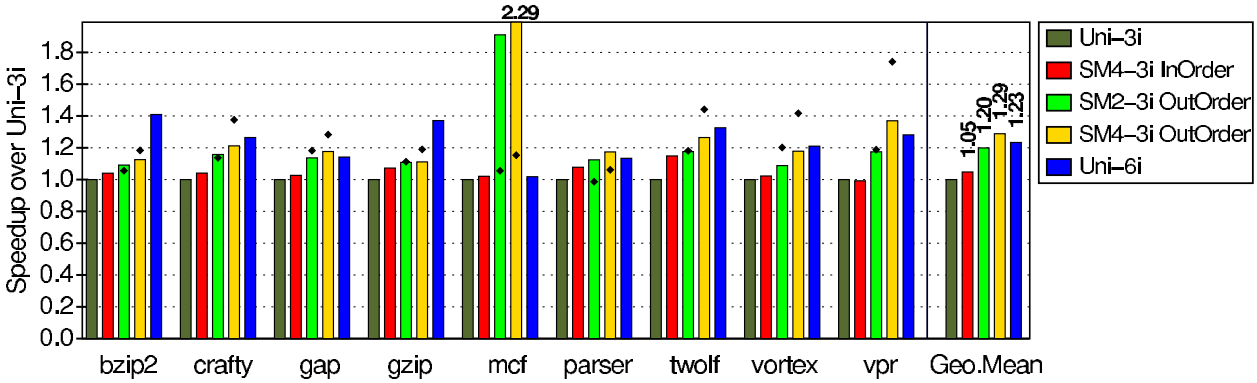

Figure 8.1: Speedups of different binary-architecture combinations relative to *BaseApp* running on *Uni-3i*. The figure also shows the geometric mean. The SM results are obtained with a fully-automated SM compiler on full SpecInt applications.

Compare first *SM4-3i*$_{OutOrder}$ to *Uni-3i*. For every single application, SM execution is faster. The speedups are always over 1.05, and reach about 2.29 for *mcf*. We will see that the *mcf* speedups are mostly due to prefetching. The geometric mean is 1.29. For the two-core SM CMP (*SM2-3i*$_{OutOrder}$) the geometric mean of the speedup is 1.2. These results make SM an attractive feature, especially given that these speedups are obtained with a *fully-automated* SM compiler, on a *decentralized* CMP architecture and, importantly, on *full* SpecInt applications.

Note that the IPC numbers in Table 8.1 do not exactly correlate with the relative height of the *Uni-3i* and *SM4-3i*$_{OutOrder}$ bars. The reason is that the binaries running on the two platforms differ.

Since one of the contributions of this thesis is support for out-of-order spawning, we compare *SM4-3i*$_{OutOrder}$ to *SM4-3i*$_{InOrder}$. The bars show that *SM4-3i*$_{InOrder}$ is much slower in all applications. *SM4-3i*$_{InOrder}$ only obtains a geometric mean speedup of 1.05 over *Uni-3i*. The magnitude of this figure is in line with previous compiler-driven SM evaluations of SpecInt2000 codes on CMPs [44], if we weight the speedups reported by the coverage of the regions that were sped up.

We conclude, therefore, that out-of-order spawn is a key enabler to boost SM speedups. The gains come from being able to exploit the additional sources of parallelism (Section 3.1).

Finally, looking at the geometric mean, we see that *SM4-3i*$_{OutOrder}$'s speedup is 6% higher than

App	IPC_{3issue}	IPC_{SM}	f_{bloat}	$f_{parallel}$	Busy CPUs	Squashed Inst. (%)	# Merges per Task Commit	Task Size (Instr)	Out Order Dyn. Inst. (%)
bzip2	1.64	1.96	1.06	1.26	1.40	10.0	0.44	743	5.6
crafty	1.42	1.82	1.06	1.45	1.97	26.2	0.29	932	38.6
gap	1.04	1.27	1.04	1.34	2.07	35.3	1.05	1270	88.1
gzip	1.05	1.25	1.07	1.28	1.49	14.1	0.03	626	0.3
mcf	0.04	0.13	1.47	1.69	2.38	28.8	0.14	47	26.3
parser	0.63	0.92	1.22	1.23	2.03	39.4	0.74	167	81.8
twolf	0.74	1.01	1.07	1.55	1.62	4.5	0.30	409	23.7
vortex	1.49	1.89	1.08	1.53	1.82	15.7	0.15	488	77.2
vpr	0.95	1.65	1.27	2.22	3.14	29.5	0.63	212	61.4
Avg	1.00	1.32	1.15	1.51	1.99	22.6	0.42	544	44.8

Table 8.1: Characterizing the run-time behavior of *SM4-3i*$_{OutOrder}$.

Uni-6i. Therefore, a SM CMP architecture compares favorably against a wider superscalar for SpecInt, even assuming that the wider superscalar cycles at no lower frequency. This is significant, given that the CMP has a natural advantage on truly parallel codes, such as many numerical applications.

8.2 Understanding SM Speedups

To understand the speedups of *SM4-3i*, we break down the execution time of *Uni-3i* and *SM4-3i*$_{OutOrder}$ into the product of committed instructions times average CPI. In the formula, CPI_{SM} corresponds to the combined CPI of all the cores in the chip.

$$Speedup_{SM} = \frac{T_{3issue}}{T_{SM}} = \frac{I_{3issue} \times CPI_{3issue}}{I_{SM} \times CPI_{SM}}$$

As we go from *Uni-3i* to *SM4-3i*$_{OutOrder}$, the number of committed instructions in a program increases. The reasons are the additional spawn, commit and memory instructions, and the lower effectiveness of conventional compiler optimizations (Section 7.3). Therefore, we define an instruction bloat factor $f_{bloat} = \frac{I_{SM}}{I_{3issue}}$.

We can put CPI_{SM} as a function of the average CPI per core, which we call CPI_{SMcore}. For that, we need to measure the time each CPU is busy executing instructions (t_i) and add it up across all the CPUs in the SM chip. The two CPIs are related as:

$$I_{SM} = \frac{T_{SM}}{CPI_{SM}} = \frac{\sum_{i=1}^{numcores} t_i}{CPI_{SMcore}}$$

Intuitively, if no two CPUs are busy at the same time, there is no parallelism, and the two

CPIs are the same. If, instead, all 4 CPUs completely overlap their busy time, parallelism is 4, and CPI_{SMcore} is 4 times CPI_{SM}. We define the parallelism factor as:

$$f_{parallel} = \frac{\sum_{i=1}^{numcores} t_i}{T_{SM}} = \frac{CPI_{SMcore}}{CPI_{SM}}$$

Consequently, the SM speedup above is:

$$Speedup_{SM} = \frac{I_{3issue} \times CPI_{3issue}}{I_{SM} \times CPI_{SM}} = \frac{f_{parallel} \times CPI_{3issue}}{f_{bloat} \times CPI_{SMcore}}$$

Table 8.1 shows the values of some of these parameters for *SM4-3i*$_{OutOrder}$ running each of the applications. Specifically, Column 4 shows the instruction bloat factor f_{bloat}. Its average value is 1.15, which indicates that SM execution increases the dynamic instruction count significantly. This effect hurts SM speedups.

Column 5 shows the parallelism factor $f_{parallel}$, which helps SM speedups. On average, its value is 1.51. $f_{parallel}$ is small because of the limited parallelism present in SpecInt codes. Note that $f_{parallel}$ reports the average number of CPUs that are busy at a given time executing tasks that will not be squashed. In reality, a higher number of CPUs is busy, but some of them execute tasks that will eventually be squashed. The true number of busy CPUs is shown in Column 6. Its average value is 1.99. We can see, therefore, that task squashing is not negligible. In fact, Column 7 shows the fraction of graduated instructions that correspond to squashed tasks. On average, such number is 22.6%. Overall, *SM4-3i*$_{OutOrder}$ wastes many cycles to squashed tasks, which also limits its speedups[1].

We can now go back to the $Speedup_{SM}$ equation and assume that $CPI_{3issue} = CPI_{SMcore}$. In this case, the SM speedups would be given by $\frac{f_{parallel}}{f_{bloat}}$. We have computed this ratio and shown it as dots in Figure 8.1 for *SM2-3i*$_{OutOrder}$ and *SM4-3i*$_{OutOrder}$.

If these dots are not equal to the real speedups, it is because $CPI_{3issue} \neq CPI_{SMcore}$. In particular, if a dot is lower than the SM bar (e.g. in parser), it means $CPI_{3issue} > CPI_{SMcore}$. This is largely due to *prefetching* effects. In particular, tasks that eventually get squashed bring data and instructions into the caches, which are later reused by other tasks. If, instead, the dot is higher than the SM bar (e.g. *vortex*), it means $CPI_{3issue} < CPI_{SMcore}$. In this case, SM execution is largely impaired by the higher average memory latency induced by cache-coherence invalidations, higher instruction cache miss rate, and slower cache hierarchy speed. It is also hurt by lower branch

[1] In all this discussion, we have only counted graduated instructions. There is an additional waste in both SM and non-SM chips caused by misspeculated branches.

predictor accuracy due to code partitioning. We call these effects SM overheads.

Figure 8.1 shows that in *SM2-3i*$_{OutOrder}$, the prefetching effect typically dominates (most of the dots are lower than the SM bars). It often adds a net 5-10% to the potential speedup from parallelism, represented by the dots. However, as we add more cores to the chip (*SM4-3i*$_{OutOrder}$), the SM overheads dominate, and often the potential speedup from parallelism is higher than the real speedup by a net 10-30%. The obvious exceptions are *mcf* and *parser*, where prefetching always dominates, and *vortex*, where the SM overheads dominate. *mcf* benefits significantly from prefetching into the L2. Its L2 miss rate decreases by 27% from *Uni-3i* to *SM4OutOrder*. *vortex* hurts from higher data and instruction L1 miss rates.

Overall, we conclude that our full SpecInt speedups are a combination of several factors. Our SM machinery is frequently able to overlap execution of the CPUs (Column 6 of Table 8.1), although a non-trivial fraction of the work is useless (Column 7). However, even after producing useful overlap (Column 5), SM needs to offset significant code bloat (Column 4) to deliver speedups. Finally, while prefetching helps SM, Figure 8.1 shows that prefetching's good effect can be overwhelmed by the opposite effects of the SM overheads.

The remaining columns of Table 8.1 further characterize *SM4-3i*$_{OutOrder}$'s execution. Column 8 shows the frequency of task merges per task commit. We can see that task merge occurs frequently in all codes. On average, there are 0.42 merges per commit. This operation boosts SM performance, as it increases task size and, as a result, reduces SM overheads.

Column 9 shows the resulting average number of graduated instructions in the tasks that commit. On average, a task contains 544 instructions.

Finally, the last column shows the percentage of committed dynamic instructions belonging to tasks spawned out of order. It varies noticeably across applications, although all codes except *bzip2* and *gzip* have a large percentage of dynamic instructions in tasks spawned out-of-order. These are the ones responsible for the speedups of *SM4-3i*$_{OutOrder}$ over *SM4-3i*$_{InOrder}$ in Figure 8.1. Generally, the fraction of dynamic instructions is correlated with the difference between *SM4-3i*$_{OutOrder}$ and *SM4-3i*$_{InOrder}$ in Figure 8.1: *crafty*, *gap*, *parser*, *twolf*, *vpr*, and *vortex* have high fractions and large differences, while *bzip2* and *gzip* have a small fraction and a small difference. *mcf* is a special case with 26.3% of the commited dynamic instructions in *mcf* spawned out-of-order and a very large

difference. If we see where the dots are for *mcf*, we realize that the speed up is largely delivered by the prefetching provided by squashed tasks that were spawned out-of-order. Overall, on average, 44.8% of the commited dynamic instructions are in tasks spawned out-of-order.

8.3 Energy Optimizations: Architectural Characterization

We start by measuring the architectural characteristics of *SM4-3i*$_{OutOrder}$ related to the SM sources of energy consumption and the energy-centric optimizations of Table 5.1. This section introduces the *Power* binary in addition to *InOrder* and *OutOrder*. As shown in Table 7.2, the *Power* binary is based on the *OutOrder* binary with the energy profiler (Section 6.3.2) activated. The data is shown in Table 8.2 and Table 8.3. In the tables, the chip before optimization is labeled *NoOpt* (*OutOrder* binary), while the chip with one optimization is labeled *SOR* (*OutOrder* binary, if tasks stall on second restart), *EProf* (*Power* binary, if it uses energy-aware task pruning by profiling), or *NoWalk* (*OutOrder* binary, if it eliminates eager tag walks).

Apps	Squashed Instructions (%)			Busy CPUs		
	NoOpt	SOR	EProf	NoOpt	SOR	EProf
bzip2	10.0	7.5	9.9	1.40	1.35	1.41
crafty	26.2	25.4	18.9	1.97	1.95	1.70
gap	35.3	31.6	35.1	2.07	1.94	2.06
gzip	14.1	14.0	11.9	1.49	1.48	1.49
mcf	28.8	28.7	28.8	2.38	2.38	2.38
parser	39.4	29.9	13.8	2.03	1.85	1.25
twolf	4.5	4.4	4.4	1.62	1.62	1.62
vortex	15.7	15.4	7.7	1.82	1.81	1.49
vpr	29.5	29.2	27.9	3.14	3.13	2.61
Avg	22.6	20.7	17.6	1.99	1.95	1.78

Table 8.2: Impact of energy-centric optimizations in the percentage of squashed instructions and busy cycles for the *SM4-3i* chip.

The first SM source of energy consumption in Table 5.1 is the work of squashed tasks. Column 2 of Table 8.2 shows that, on average, *NoOpt* loses to task squashes 22.6% of the dynamic instructions executed. This is a significant waste. With our optimizations, we reduce the fraction of such instructions. Specifically, the average fraction becomes 20.7% with *SOR* (Column 3) and 17.6% with *EProf* (Column 4). Although not shown in the table, the fraction becomes 16.9% with both

Apps	Pruned Tasks (%)	Task Size (Instructions)		ED^2 Reduc. (%)	Ratio of Tag Accesses (SM/*Uni-3i*)		Traffic (SM/ *Uni-3i*)	Add'l Graduated and Committed Instructions (%)	
	EProf	NoOpt	EProf	EProf	NoOpt	NoWalk	NoOpt	NoOpt	EProf
bzip2	21.4	743	751	-0.3	3.2	1.3	2.5	5.6	5.6
crafty	5.8	932	1064	6.8	2.9	2.0	3.6	5.6	5.6
gap	14.2	1270	1280	-0.8	3.6	2.2	8.4	3.8	3.8
gzip	7.1	626	634	0.3	3.5	1.9	4.0	6.5	6.5
mcf	0.0	47	47	0.0	3.8	2.7	11.5	31.9	31.9
parser	18.5	167	261	26.4	3.6	3.2	7.1	20.8	18.0
twolf	0.0	409	409	0.0	3.3	1.6	3.2	6.5	6.5
vortex	8.8	488	881	16.0	2.9	1.9	3.9	7.5	7.4
vpr	16.6	212	389	10.4	3.2	3.1	6.4	23.9	21.2
Avg	10.3	544	635	6.5	3.3	2.2	5.6	12.5	11.9

Table 8.3: Architectural characteristics of the *SM4-3i* chip related to SM sources of energy consumption and their optimization.

optimizations combined.

The next few columns of Table 8.2 provide more information on the impact of *SOR* and *EProf*. Under *NoOpt*, the average number of busy CPUs is 1.99 (Column 5). Since *SOR* stalls tasks that may be squashed and *EProf* removes them, they both reduce CPU utilization. Specifically, the average number of busy CPUs is 1.95 and 1.78 with *SOR* and *EProf*, respectively (Columns 6 and 7). With both optimizations, the average can be shown to be 1.75.

Table 8.3 provides additional insights on the impact of energy-centric optimizations. *EProf* has a significant impact on the tasks. On average, it prunes 10.3% of the static tasks (Column 2), increasing the average task size from 544 instructions in *NoOpt* (Column 3) [2] to 635 (Column 4). Moreover, the average $E \times D^2$ product of the applications decreases by 6.5% (Column 5).

The next SM source of energy in Table 5.1 is dominated by accesses to tags in the cache hierarchy that contain LIDs or global task IDs (Figure 5.3). In *Uni-3i*, every load and store requires an L1 tag check and, if it misses, additional L2 and L1 tag checks. Under SM, these tag checks may consume slightly more energy because they may include ID checks. Moreover, background FSMs periodically walk the L1 tags, inducing more checks. With *NoWalk*, we eliminate most of these background checks. Column 6 of Table 8.3 shows that, on average, *NoOpt* has 3.3 times the number of tag checks in *Uni-3i*. With *NoWalk*, this number is reduced to 2.2 (Column 7). Note that these figures include the contribution of squashed tasks.

[2] This is the same value as Table 8.1 column 9

The next SM source of energy is additional traffic. Column 8 of Table 8.3 shows that, on average, *NoOpt* has 5.6 times the traffic of *Uni-3i*. To compute the traffic, we add up all the bytes of data or control passed between caches. This traffic increase is caused by the factors described in Section 4.3, including parallelization of the code. After we apply our optimizations, since they eliminate some squash and other effects, the traffic reduces to 4.3 times that in *Uni-3i* (not shown in the table).

The final SM source of energy is additional instructions. Column 9 shows that *NoOpt* executes on average 12.5% more graduated and committed instructions than *Uni-3i*. The *EProf* optimization, by eliminating small, inefficient tasks, reduces the additional instructions to 11.9% on average (Column 10).

8.4 The Energy Cost of SM (ΔE_{SM})

We define the energy cost of SM (ΔE_{SM}) as the difference between the energy consumed by our SM CMPs and *Uni-3i*. Our goal is to quantify ΔE_{SM} and its main components with and without our optimizations.

Figure 8.2 characterizes ΔE_{SM} for *SM4-3i*. The figure shows five bars for each application. They correspond to the total energy consumed by the chip without any optimization (*NoOpt*), with individual optimizations applied (*SOR*, *EProf*, and *NoWalk*), and with all optimizations applied (*SM4-3i*$_{Power}$). For each application, the bars are normalized to the energy consumed by *Uni-3i*. Consequently, the *difference between the top of the bars and 1.00* is ΔE_{SM}.

For each bar, Figure 8.2 also shows the contributions of the main SM-specific sources of energy consumption listed in Table 5.1. These include the total energy consumed by squashed tasks (ΔE_{Squash}), by storage and logic for data versioning ($\Delta E_{Version}$), by the additional traffic ($\Delta E_{Traffic}$), and by the additional graduated and committed instructions (ΔE_{Inst}). The rest of the bar (*Non-SM*) is energy not directly attributable to SM, and will be considered later.

Ideally, ΔE_{SM} should be roughly equal to the addition of the four SM-specific sources of energy consumption and, therefore, *Non-SM* should equal 1. In practice, this is not the case because a given program runs on *SM4-3i*$_{Power}$ and *Uni-3i* at different speeds and temperatures. As a result, the remaining dynamic and leakage energy necessarily changes between runs, causing *Non-SM* to

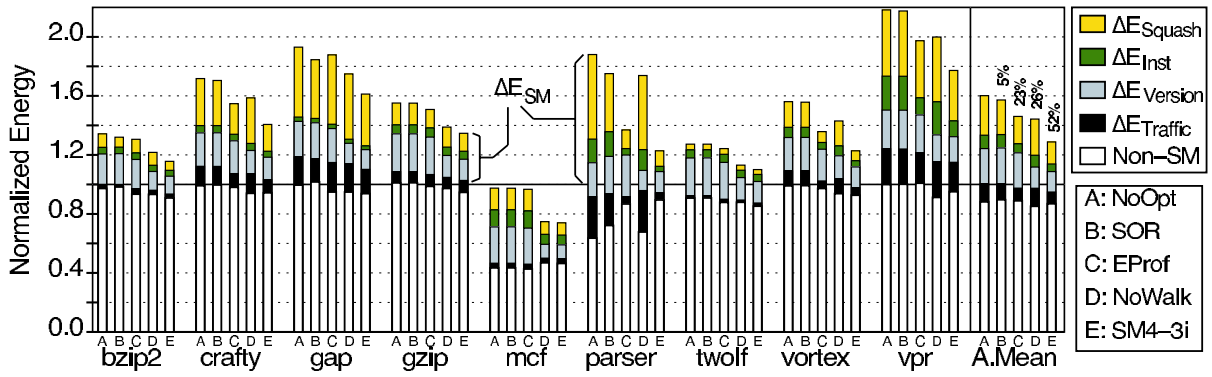

Figure 8.2: Assessing the energy cost of SM for the *SM4-3i* chip with and without energy-centric optimizations. The percentages listed above the average bars are the decrease in the energy cost of SM (ΔE_{SM}) when the optimization is activated.

deviate from 1. In fact, since for all applications *SM4-3i*Power is faster than *Uni-3i* (Section 8.7), *Non-SM* is less than 1: non-SM hardware structures have less time to leak or to spend dynamic energy cycling idly.

If we consider the *NoOpt* SM environment, we see that the energy cost of *unoptimized* SM (ΔE_{SM}) is significant. On average for our applications, unoptimized SM adds 60% to the energy consumed by *Uni-3i*. We also see that, of our SM sources of energy consumption, two dominate, namely energy in squashed tasks and in structures for data versioning. On average, together they represent over 75% of SM-specific consumption.

Since we are not interested in the *NoOpt* environment, we do not analyze it further.

8.5 The Impact of Energy-Centric Optimizations

We now consider the rest of the bars in Figure 8.2. Recall that our optimizations target SM energy sources. Consequently, their effectiveness is roughly (but not strictly) bound by the energy cost of SM (ΔE_{SM}), which is the energy over the horizontal line at 1.00 in the figure. With this in mind, observe that our optimizations are indeed effective. On average, *EProf* eliminates 23% of ΔE_{SM}, *NoWalk* 26%, and *SOR* a modest 5%. While these optimizations are not perfectly additive, the combination of all three eliminates on average 52% of ΔE_{SM}. Compared to the overall energy consumed by *NoOpt*, this is a very respectable 20% energy reduction with practically no slowdown.

The figure shows that the optimizations reduce the sources of energy consumption that they

are expected to minimize from Table 5.1. This is better seen from the average bars. Consider *EProf* first. In Figure 8.2, *EProf* reduces ΔE_{Squash} and ΔE_{Inst}. This is expected from Table 8.2, where *EProf* reduces the squashed instructions from 22.6% to 17.6%, and decreases the additional graduated and committed instructions from 12.5% to 11.9%. In addition, since *EProf* reduces squashing, it also indirectly reduces $\Delta E_{Traffic}$ in Figure 8.2. This indirect effect was discussed in Section 8.3.

In Figure 8.2, *NoWalk* reduces $\Delta E_{Version}$. This is expected from Table 8.3, where *NoWalk* reduces the ratio of tag accesses from 3.3 to 2.2. In addition, since it reduces the temperature, it also reduces the leakage component in *Non-SM* slightly. Finally, *SOR* only addresses ΔE_{Squash}. As expected from the modest numbers in Table 8.2, where it reduces squashed instructions from 22.6% to 20.7%, it has a small impact in Figure 8.2.

When we combine all these optimizations in *SM4-3i*, we see that all four SM sources of consumption decrease. The resulting *SM4-3i* bar shows the *true energy cost* of SM. If we measure the section of the bar over 1.00, we see that this cost is on average only 28%. We feel that this is a remarkably low overhead for SM.

If we examine individual applications, we see that ΔE_{SM} after the optimizations is typically 10-60%. The two outliers are *mcf* and *vpr*, which show positive and negative effects of SM. In *mcf*, the L2 suffers frequent misses without SM; with SM, threads prefetch data for other threads, removing misses and speeding up the execution significantly (Section 8.7). As a result, ΔE_{SM} is negative (-26%). In *vpr*, the distribution of computation across cores in SM hurts L1 cache locality and branch predictor accuracy. The result is a lower L1 hit rate. While the program still runs faster under SM, the result is a high ΔE_{SM} (78%).

Finally, the per-application contributions of ΔE_{Squash}, ΔE_{Inst}, $\Delta E_{Version}$, and $\Delta E_{Traffic}$ in Figure 8.2 are generally well-correlated with data on squashed instructions, additional instructions, ratio of tag accesses, and traffic, respectively (shown in Table 8.2 and Table 8.3) One notable exception is $\Delta E_{Traffic}$ in *mcf* where, due to frequent processor stall, $\Delta E_{Traffic}$ accounts for a small fraction of the total energy despite the large traffic increase in SM. Also, note that *parser* increases *Non-SM* noticeably after applying *EProf*. The reason is that, by pruning tasks, *EProf* slows down the program by about 7%, which has the effect of boosting *Non-SM*. This is the only

case where our optimizations significantly affect the execution time of the application.

8.6 Energy Consumption Breakdown

To further characterize the optimized *SM4-3i$_{Power}$*, we compare its energy consumption to the wider superscalar (*Uni-6i*). Figure 8.3 shows the energy consumed by *SM4-3i$_{Power}$*, *Uni-6i* and, for completeness, *Uni-3i* and *SM2-3i$_{Power}$*. Each bar is normalized to *Uni-3i* and broken down into dynamic energy consumed by the core, clock, and memory subsystem, and leakage energy. The memory category includes caches, TLBs, and interconnect.

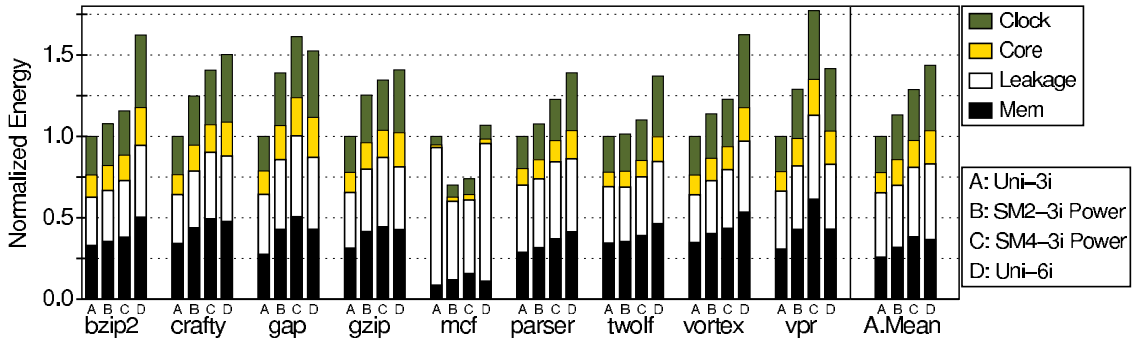

Figure 8.3: Comparing the energy consumption. The bars are normalized to *Uni-3i*.

The breakdown gives insight into *Uni-6i*'s consumption. Its core, clock, and memory categories are larger than in *Uni-3i* because of the bigger structures in the wide processor. Specifically, the rename table, register file, I-window, L1 data cache, and data TLB have twice the number of ports. This roughly doubles the energy per access [30]. Furthermore, all these structures but the cache and TLB also have more entries. Finally, the forwarding bus also increases its complexity and, therefore, its consumption. The figure also shows that leakage has increased. The reason is that, while *Uni-6i* is fast, it consumes the highest average power (Section 8.7) and, therefore, has high temperature. Recall that temperature has an exponential impact on leakage.

Compared to *Uni-6i*, *SM4-3i$_{Power}$* has smaller core and clock energies because it has simpler structures. Its leakage is smaller because its average power (Section 8.7) and, therefore, temperature are smaller than *Uni-6i*. Its memory category, however, is about the same because of the data versioning support. Overall, *SM4-3i$_{Power}$* consumes on average 17% less energy than *Uni-6i*.

SM2-3i$_{Power}$'s consumption is between that of *Uni-3i* and *SM4-3i$_{Power}$*.

8.7 Performance and Power Evaluation

Finally, we compare *SM4-3i$_{Power}$*'s performance and average power to *Uni-6i*'s. Figure 8.4-(a) shows application speedups relative to execution on *Uni-3i*, while Figure 8.4-(b) shows the average power consumed during execution. As usual, *Uni-3i* and *SM4-3i$_{Power}$* are also shown.

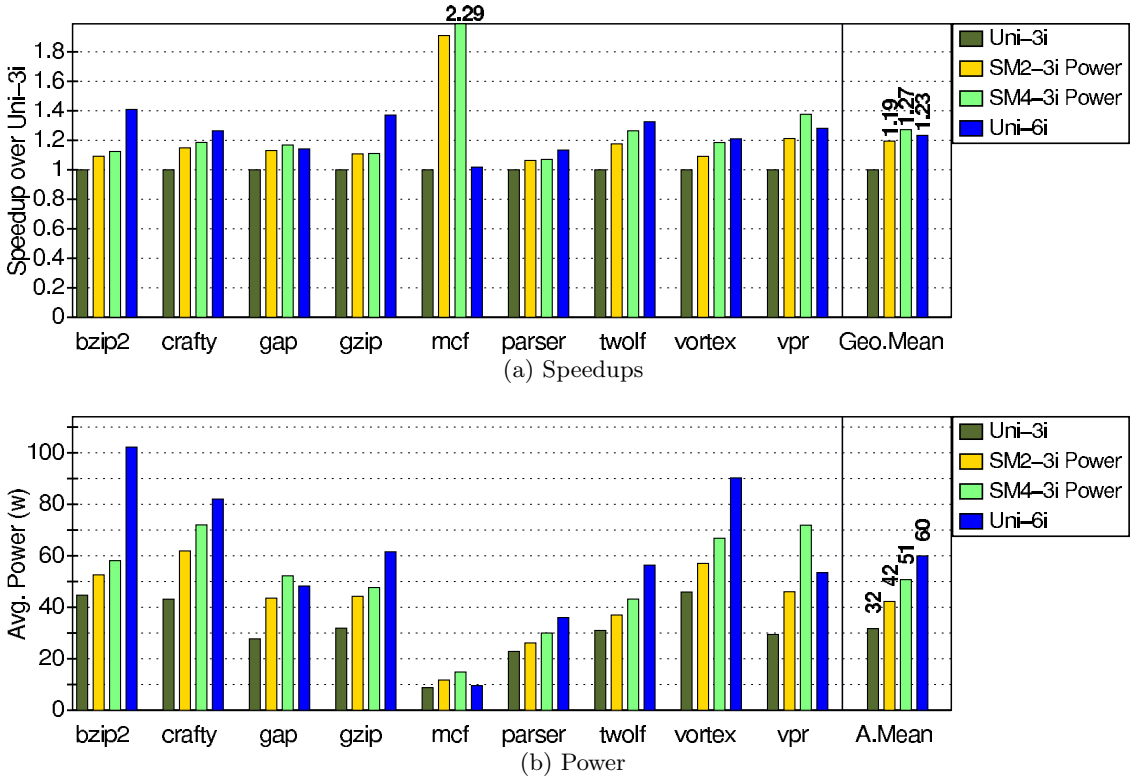

Figure 8.4: Execution speedup relative to *Uni-3i* (a) and average power consumption (b) for different chip organizations.

Figure 8.4-(a) is very similar to Figure 8.1. The key difference is that the *Power* binary is used instead of the *OutOrder* and *InOrder* binaries. It shows that, on average, *SM4-3i$_{Power}$* delivers a speedup of 1.27 over *Uni-3i*. This shows that our SM compiler successfully extracts good tasks from these irregular codes even when the energy profiler is activated. While the speedup for most codes ranges from 1.10 to 1.35, *mcf* exhibits a higher speedup. As indicated in Section 8.5, *mcf* benefits from constructive data prefetching into L2 by SM tasks.

The figure also shows that *SM4-3i$_{Power}$* is still on average slightly faster than *Uni-6i*. The average speedup for *SM4-3i$_{OutOrder}$* is 1.29, therefore the performance degradation when all the energy-centric optimizations are activated is 2%. The speculative parallelism enabled by *SM4-*

3i$_{Power}$ in these hard-to-parallelize codes is more effective than doubling the issue width. This is a good result, especially because we conservatively assume the same frequency for both chips. In practice, designing the wider issue processor at this high frequency is likely to be more challenging.

On the other hand, Figure 8.4-(b) shows that the average power consumed by *SM4-3i*$_{Power}$ is typically lower than *Uni-6i*'s. On average, it is 15% lower. Moreover, it never reaches the high values that *Uni-6i* dissipates in some applications. To summarize, *SM4-3i*$_{Power}$ is significantly more energy-efficient than *Uni-6i*: it is slightly faster and consumes 15% less power.

We can get further insight if we analytically apply *ideal* voltage-frequency scaling. We assume that performance is linearly proportional to frequency and scale frequency and voltage proportionally. We also assume that average dynamic power is proportional to the cube of frequency and that average leakage power is linearly proportional to voltage [6]. Then, for each chip, we can derive a curve that relates the average power consumption with performance as:

$$P_{new}^{total} = P_{orig}^{dyn} \times \left(\frac{Speedup_{new}}{Speedup_{orig}}\right)^3 + P_{orig}^{leak} \times \left(\frac{Speedup_{new}}{Speedup_{orig}}\right)$$

Figure 8.5 shows the resulting curves for *SM4-3i*$_{Power}$ and *Uni-6i*. Each curve follows possible speedup-power working points for one chip. Each curve shows a data point, which corresponds to the actual working conditions of the chip. The lower a curve is, the more energy-efficient the architecture is.

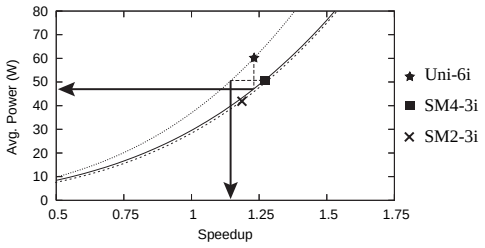

Figure 8.5: *Ideal* relation between speedup and average power.

We can see that *Uni-6i* is less energy-efficient than *SM4-3i*$_{Power}$. If we scale down *SM4-3i*$_{Power}$'s frequency until *SM4-3i*$_{Power}$'s performance is equal to *Uni-6i*'s, *SM4-3i*$_{Power}$ consumes 20% less power than *Uni-6i* (horizontal arrow). Alternatively, if we scale down *Uni-6i*'s frequency until *Uni-6i*'s power is equal to *SM4-3i*$_{Power}$'s, *SM4-3i*$_{Power}$ is 13% faster than *Uni-6i* (vertical arrow).

Finally, Figure 8.5 also shows a curve for *SM2-3i*$_{Power}$. As expected due to the smaller SM overhead, the data shows that *SM2-3i*$_{Power}$ has a small efficiency advantage over *SM4-3i*$_{Power}$.

Chapter 9

Related Work

9.1 Out-of-Order Spawning

Hammond *et al.* [16] propose a SM CMP where each processor has a co-processor that controls SM mechanisms with software handlers. They support both subroutine and loop-iteration tasks and, therefore, out-of-order spawn. Co-processors are told what task is running where. They snoop on two broadcast buses and, based on message source, they can tell the relative ordering. Since caches contain state from a single task, no task ID is necessary. Squash signals are also broadcast. Commits require access to a centralized software data structure in shared memory. The most speculative task is killed if there is no space in the CMP. Overall, this is a broadcast-based, relatively centralized architecture. The authors conclude that their scheme has too much software overhead to support subroutine tasks. Their findings motivate our search for hardware-based mechanisms.

There are several high-level performance-evaluation studies of environments that need out-of-order spawn [24, 26, 42, 43]. They often assume some ideal architectural feature, such as an infinite number of processors or perfect value prediction, and compare the performance to more realistic environments. Of those, [24, 26] examine a variety of sources of parallelism, including iterations from multiple loop levels and nested subroutines. [42, 43] examine subroutine-level nested parallelism. None of these papers attempts to describe microarchitectural structures to support the tasks used. Consequently, they have not addressed the problems we cover. Our paper is the first detailed microarchitectural design of high-speed out-of-order tasking on a CMP.

DMT is a centralized, SMT-like processor whose hardware can extract out-of-order tasks from unmodified binaries [2]. The design uses centralized structures that are *unusable* in a CMP. Specif-

ically, DMT has a centralized hardware tree that records which tasks are successors of which. To determine the order of two tasks, the hardware walks the tree when: (1) there is a collision in the centralized LD/ST queue, or (ii) a task commits and needs to verify the register predictions for successor tasks. This centralization means that DMT does not need our proposed IS list and timestamp intervals. DMT kills the most speculative task if there is no space in the processor, while we use task merge to dynamically manage the resources in the system.

Dubey *et al.*'s SPSM [13] is an architecture where tasks are spawned in order. Interestingly, a task can spawn multiple other tasks, but these other tasks cannot further spawn, which guarantees in-order spawn. Our proposed spawning model is more flexible and enables more parallelism.

Littin *et al.*'s WarpEngine [29] is a compute engine where instructions are grouped into 16-instruction branch-less frames. Frames are fetched and executed out of order. The machine appears closer to an aggressive dynamic superscalar that exploits control continuations. For example, it cannot be used as a multiprocessor for parallel applications.

In Multiscalar [31], a task may have multiple exit points. However, only one is correct. Since a task can only spawn a single other *correct* task in its lifetime, Multiscalar supports in-order spawn only.

9.1.1 Related Mechanisms: Timestamping and Merging

Hood *et al.* [17] track on-the-fly data references to detect for anomaly detection, their implementation also can have accessed performed out-of-order. They proposed a software-only solution where induction variable is used to induce the relative order.

Cleary *et al.* [12] propose several timestamp representations for virtual sequences organized in a tree. They bear some resemblance to our splitting timestamp interval. However, while some of Cleary *et al.*'s schemes are more efficient than others, they all need periodic *re-scaling*. Re-scaling occurs when sequences run out of timestamps. In that case, new timestamps need to be reassigned to *all the tasks* on the fly. This is a very costly operation, which would entail synchronizing the whole machine, and walking all the cache tags, changing all the timestamps. Our splitting timestamp interval scheme is designed for *efficient hardware implementation*. Once a base timestamp is assigned to a task, it never changes. The scheme does not need re-scaling. Thanks to

the support for automatic dynamic timestamp expansion (Section 5.4) and timestamp wrap around (Section 5.4.1), we practically never have to kill a task.

Dubey *et al.*'s SPSM [13] can perform conditional spawns. This is somewhat similar to our dynamic task merging. A key difference is that their mechanism does not allow a parent who initiated a merge to pass the responsibility of completing the merge to a child. Moreover, their mechanism works with in-order spawn only, while ours is for out-of-order spawn, which increases complexity. In addition, in SPSM only the safe task can perform conditional spawn, while in our mechanism any task can perform task merging. Overall, our mechanism is more flexible. However, it needs a NES counter per task, which may be passed between tasks.

Multiscalar [31] introduces the concept of suppress register. When a task suppresses a section of code (typically a function) the task ignores all the Multiscalar instrumentation in the code. In addition, it increments a counter. Suppressions can be nested, in which case the counter keeps increasing. However, the task *cannot spawn* a successor until all its (nested) suppressed sections are completed and the counter reaches zero. This optimization is typically used to avoid code replication. Our mechanism for dynamic task merging is more flexible. A task can start a merge operation and then, as load conditions in the machine change, decide to spawn successors that will complete the merge. This is done by passing the parent's NES counter to its successors. As a result, dynamic task merging is a powerful tool to manage dynamically-changing resources efficiently.

Park *et al.* [27] propose multiplexing a number of in-program-order threads into a single hardware context of IMT. This is very different from our dynamic task merging. In IMT, each of the threads multiplexed in a context still keeps a PC and a rename table pointer. The technique appears similar to recursively applying SMT to each hardware context of a SMT. Our proposal keeps a single PC and a single set of architectural registers for all the merged tasks.

9.2 Energy Considerations in Speculative Multithreading

Past work on CMP architectures with SM support has focused on performance rather than energy (e.g., [8, 15, 16, 19, 23, 31, 32, 36, 37, 46]). There has been work on reducing the energy consumed in the pipeline due to instruction-level speculation following a branch prediction [3, 22]. However, the issues addressed are very different.

Chapter 10

Conclusions

This thesis has two sets of contributions: Energy and Performance.

To achieve better performance than previously proposed SM CMP systems, this thesis has been the first to identify and design a set of microarchitectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a SM CMP. The three mechanisms are Splitting Timestamp Intervals, Immediate Successor List, and Dynamic Task Merging. They address the two main challenges posed by out-of-order spawning: correct and efficient task ordering and resource allocation.

This thesis also challenges the commonly-held view that SM consumes excessive energy. It shows that it is possible to design a SM CMP that is energy efficient. To do so, it identifies the main sources of energy consumption in SM and proposes energy-centric optimizations to mitigate them. These sources are squashed tasks, storage and logic to support data versioning, additional traffic, and additional instructions. The optimizations eliminate 20% of the energy consumption in a SM CMP without noticeable performance impact. This cuts by half what we called the energy cost of SM.

With this support and our fully-automated SM compiler for out-of-order spawn, we unlock the potential of SM for hard-to-speedup integer codes. Specifically, a SM CMP with 4 3-issue cores delivers an average speedup of 1.27 for *full* SpecInt 2000 applications; without out-of-order spawn, we obtain an average speedup of 1.05, in line with past SM CMP work on the same codes. Although the proposed architecture has approximately the same area that a 6-issue processor, it is not only faster but also more energy-efficient. Applying ideal frequency scaling, for the same average power in both chips, the SM CMP is 13% faster than the superscalar.

These results make SM a compelling feature, given that they are obtained with a *fully-automated* SM compiler, on a *decentralized* CMP architecture and, importantly, on *full* SpecInt applications. Moreover, we feel that these results can be improved, as the opportunities for speculative multi-threading in these most challenging applications become better understood.

CMPs are attractive because they are more energy-efficient, more scalable, and have lower complexity than wide-issue superscalars. Moreover, they have an advantage for explicitly-parallel codes. The thesis showed that SM CMPs can also speed up the challenging SpecInt codes, and deliver a slightly better performance than wider superscalars, for a smaller energy budget.

References

[1] *International Technology Roadmap for Semiconductors*. Semiconductor Industry Association, 2002.

[2] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, November 1998.

[3] J. L. Aragon, J. Gonzalez, and A. Gonzalez. Power-Aware Control Speculation Through Selective Throttling. In *Proceedings of the 9th High-Performance Computer Architecture Conference*, pages 103–112, February 2003.

[4] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] J. Adam Butts and Gurindar S. Sohi. A static power model for architects. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 191–201, 2000.

[7] M. Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[8] M. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[9] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. In *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 25–36, June 2003.

[10] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, June 2000.

[11] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th High-Performance Computer Architecture Conference*, February 2002.

[12] J.G. Cleary, J.A.D. McWha, and M.W. Pearson. Timestamp representations for virtual sequences. In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 98–105, 1997.

[13] P. Dubey, K. O'Brien, and C. Barton. Single-Program Speculative Multithreading (SPSM) Architecture. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, 1995.

[14] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *Proceedings of the 9th High-Performance Computer Architecture Conference*, pages 191–202, February 2003.

[15] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[16] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[17] R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detecction. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 74 – 81, 1990.

[18] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *In Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[19] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[20] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, December 2003.

[21] X. F. Li, Z. H. Dui, Q. Y. Zhao, and T. F. Ngai. Software Value Prediction for Speculative Parallel Threaded Computations. In *First Value Prediction Workshop*, pages 18–25, June 2003.

[22] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[23] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 365–372, June 1999.

[24] P. Marcuello and A. Gonzalez. A Quantitative Assessment of Thread-level Speculation Techniques. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 595–604, 2000.

[25] Diego Novillo. Design and implementation of the treessa. In *Proceedings of the GCC Developer's Summit*, June 2004.

[26] J. T. Oplinger, D. L. Heine, and M. S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[27] I. Park, B. Falsafi, and T. Vijaykumar. Implicitly Multithreaded Processors. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[28] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA'01)*, pages 204–215, June 2001.

[29] R.H.Littin, J.A.D. McWha, M.W.Pearson, and J.G.Cleary. Block based execution and task level parallelism. In *Australian Computer Science Communications*, pages 57–66, 1998.

[30] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical Report 2001/2, Compaq Computer Corporation, August 2001.

[31] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[32] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[33] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proceedings of the 8th High-Performance Computer Architecture Conference*, February 2002.

[34] J.G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical report, CMU-CS-97-188, Carnegie Mellon University, November 1997.

[35] H. Su, F. Liu, A. Devgan, E. Acar, and S. Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, August 2003.

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

[37] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[38] J. Y. Tsai, Z. Jiang, and P. C. Yew. Compiler Techniques for the Superthreaded Architecture. In *International Journal of Parallel Programming*, pages 27(1):1–19, 1999.

[39] J. Tuck. A novel compiler framework for a chip-multiprocessor architecture with thread-level speculation. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[40] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[41] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002.

[42] F. Warg and P.Stenström. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[43] F. Warg and P. Stenström. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[44] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[45] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, Univ. of Virginia Dept. of Computer Science, March 2003.

[46] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 65–77, November 2002.

Curriculum Vitae

Personal Information

Jose Renau
Jack Baskin School of Engineering
University of California, Santa Cruz
1156 High Street
Santa Cruz, CA 95064

Citizenship Spain
Email renau@soe.ucsc.edu
Homepage www.soe.ucsc.edu/%7erenau
Phone (217) 721-5255 (mobile)
(831) 459-4829 (fax)

Research Interests

Computer architecture, chip multiprocessors, energy/performance trade-offs, thread level speculation, interaction between architecture and compilers, Linux kernel.

Education

University of Illinois at Urbana Champaign: (Advisor: Professor Josep Torrellas)

2004 Ph.D. Computer Science

Thesis: *"Chip Multiprocessor with Speculative Multithreading: Design for Performance and Energy Efficiency"*

The thesis challenges, for the first time, the commonly-held view that Speculative Multithreading (SM) consumes excessive energy. It also proposes novel micro-architectural mechanisms to support out-of-order task spawning in Chip Multiprocessors (CMP) with SM. The experimental work included the development of a full SM compiler.

1999 M.S. Computer Science

Thesis: *"Memory Hierarchies in Intelligent Memories: Energy/Performance Design"*

The thesis describes the FlexRAM architecture, focusing on energy, performance, and complexity issues. FlexRAM is a processor-in-memory architecture.

Ramon Llull University, Spain:

1997 M.S. Computer Science

Thesis: *"Linux Kernel IEEE1284 Implementation"*

The thesis consisted of building TCP/IP over IEEE1284 and SCSI in Linux. The implementation also included drivers.

1994 B.S. Computer Science

Final project: *"ILZR, a New Data Compression Algorithm"*

Awards

- IBM Graduate Research Fellowship (2003-2004)
- J. Poppelbaum Memorial Award, University of Illinois (2003). Given to one graduate student every year for academic merit and creativity in computer architecture

Publications

Conferences and Journals

- *"Managing Multiple Low-Power Adaptation Techniques: The Positional Approach"*, Michael Huang, Jose Renau, Josep Torrellas, Sidebar, IEEE **Computer Magazine**, December 2003.
- *"Programming the FlexRAM Parallel Intelligent Memory System"*, Basilio Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas, International Symposium on Principles and Practice of Parallel Programming (**PPoPP**), June 2003.
- *"Positional Adaptation of Processors: Application to Energy Reduction"*, Michael Huang, Jose Renau, and Josep Torrellas, International Symposium on Computer Architecture (**ISCA**), June 2003.

- *"Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors"*, José F. Martínez, Jose Renau, Michael Huang, Milos Prvulovic, and Josep Torrellas, International Symposium on Microarchitecture (**MICRO**), November 2002.

- *"Energy-Efficient Hybrid Wakeup Logic"*, Michael Huang, Jose Renau, and Josep Torrellas, International Symposium on Low Power Electronics and Design (**ISLPED**), August 2002.

- *"A Framework for Dynamic Energy Efficiency and Temperature Management"*, Wei Huang, Jose Renau, and Josep Torrellas, Journal on Instruction Level Parallelism (**JILP**), October 2001.

- *"Cache Decomposition for Energy-Efficient Processors"*, Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas, International Symposium on Low Power Electronics and Design (**ISLPED**), August 2001.

- *"A Framework for Dynamic Energy Efficiency and Temperature Management"*, Wei Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas, International Symposium on Microarchitecture (**MICRO**), December 2000.

Workshops

- *"Profile-Based Energy Reduction for High Performance"*, Wei Huang, Jose Renau, and Josep Torrellas, ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO), December 2001.

- *"Energy/Performance Design of Memory Hierarchies for Processor-In-Memory Chips"*, Wei Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas, Workshop on Intelligent Memory Systems, November 2000. It also appeared in Lecture Notes in Computer Science (Vol. 2107) by Springer-Verlag, 2001.

- *"Memory Hierarchies in Intelligent Memories: Energy/Performance Design"*, Wei Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas, Ninth Workshop on Scalable Shared Memory Multiprocessors, June 2000.

Technical Reports and Theses

- *"CFlex: A Programming Language for the FlexRAM Intelligent Memory Architecture"*, Basilio Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas, Technical Report UIUCDCS-R-2002-2287, July 2002.
- *"FlexRAM Architecture Design Parameters"*, Seung-Moon Yoo, Jose Renau, Wei Huang, and Josep Torrellas, Technical Report 1584, October 2000.
- *"Memory Hierarchies in Intelligent Memories: Energy/Performance Design"*, Jose Renau, M.S. Thesis, University of Illinois, December 1999.
- *"Linux Kernel IEEE1284 Implementation"*, Jose Renau, M.S. Thesis, Ramon Llull University, June 1997.

Talks

As Presenter at Conferences/Workshops

- *"Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors"*, International Symposium on Microarchitecture (MICRO), November 2002.
- *"Cache Decomposition for Energy-Efficient Processors"*, International Symposium on Low Power Electronics and Design (ISLPED), August 2001.
- *"Memory Hierarchies in Intelligent Memories: Energy/Performance Design"*, The Ninth Workshop on Scalable Shared Memory Multiprocessors, June 2000.

As Invited Speaker

- *"Architectural Support for Hierarchical Thread-Level Speculation"*, IBM T.J.Watson Research Center, New York, August 2003.

As Presenter in DARPA PI Meeting

- *"Morphable Multithreaded Memory Tiles (M3T) Architecture"*, IBM T.J.Watson Research Center, New York, April 2002.

Software Created

- Designed and implemented a new simulator of computer architectures (Sesc). It is used by several research groups at the University of Illinois, University of Rochester, North Carolina State University, Georgia Institute of Technology, and Cornell University. It models a variety of architectures, including dynamic superscalar processors, CMPs, processor-in-memory, and TLS architectures.

- Created a fully automatic TLS compiler pass using GCC. It generates tasks with software value prediction. This is the compiler used to evaluate the architecture proposed in my Ph.D. thesis.

- Made some extensions to CACTI, a widely used cache power model. The extensions have been used at the University of Illinois, University of Rochester, North Carolina State University, U.C. Davis, U.C. Irvine, U.C. Riverside, and University of Arizona.

- Contributed to official Shared Memory Multiprocessors (SMP) Linux patches to support SMP boards. These patches are included in all the Linux kernel distributions since 1995.

- Co-developed the IEEE 1284 (parallel port) in Linux. This implementation is included in all Linux kernels since 1996.

- Developed official GCC patches, which are included in the main distribution (2002).

- Developed TCP/IP over SCSI boards, which involved several modifications to the Linux kernel to support a high-performance interconnection system between Linux machines.

- Invented a new data compression algorithm (ILZR), a variant of Lempel Zib Ross William, distributed as public domain for Amiga Computers in Aminet-CD (1993).

- Developed the superscalar simulation infrastructure used by the Architecture Group at the Computer Science Department of Ramon Llull University (1992-1994).

Teaching Experience

- Substitute teacher for some senior- and graduate-level computer architecture classes at the University of Illinois (2002, 2003).
- Tutoring graduate students at the University of Illinois (2002, 2003).
- Created and taught a course for system administrators at Ramon Llull University, Spain. The course was 4 hours a week for 10 weeks (1997).

Professional Experience

Jan 1999-Aug 2003 Research Assistant. University of Illinois at Urbana-Champaign.

Aug 1998-Dec 1998 System Administrator. University of Illinois at Urbana-Champaign.
Worked for the Computing and Communications Services Office.

Jan 1998-Jul 1998 Computer Network Specialist. FIHOCA, S.A. (Spain).

Sep 1996-Sep 1997 System Administrator. Asertel, S.A. (Spain).
In charge of the computer infrastructure. Specialized in network security.

May 1995-Sep 1996 Systems Manager. Ramon Lull University (Spain).
In charge of the administration of the UNIX machines, PCs, and the network of the University.

Profesional Activities and Memberships

- Reviewer of papers for conferences and journals in computer architecture (ISCA, MICRO, HPCA, ICS, CAL, and IPDPS).
- ACM member since 1997.