Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2009

Abstract

The current trend toward chip multiprocessor architectures has placed great pressure on programmers and compilers to generate thread-parallel programs. Improved execution performance can no longer be obtained via traditional single-thread instruction level parallelism (ILP), but, instead, via multithreaded execution. One notable technique that facilitates the extraction of parallel threads from sequential applications is thread-level speculation (TLS). This technique allows programmers/compilers to generate threads without checking for inter-thread data and control dependences, which are then transparently enforced by the hardware. Most prior work on TLS has concentrated on thread selection and mechanisms to efficiently support the main TLS operations, such as squashes, data versioning, and commits.

This thesis seeks to enhance TLS functionality by combining it with other speculative multithreaded execution models. The main idea is that TLS already requires extensive hardware support, which when slightly augmented can accommodate other speculative multithreaded techniques. Recognizing that for different applications, or even program phases, the application bottlenecks may be different, it is reasonable to assume that the more versatile a system is, the more efficiently it will be able to execute the given program.

As mentioned above, generating thread-parallel programs is hard and TLS has been suggested as an execution model that can speculatively exploit thread-level parallelism (TLP) even when thread independence cannot be guaranteed by the programmer/compiler. Alternatively, the helper threads (HT) execution model has been proposed where subordinate threads are executed in parallel with a main thread in order to improve the execution efficiency (i.e., ILP) of the latter. Yet another execution model, runahead execution (RA), has also been proposed where subordinate versions of the main thread are dynamically created especially to cope with long-latency operations, again with the aim of improving the execution efficiency of the main thread (ILP).

Each one of these multithreaded execution models works best for different applications and application phases. We combine these three models into a single execution model and single hardware infrastructure such that the system can dynamically adapt to find the most appropriate multithreaded execution model. More specifically, TLS

is favored whenever successful parallel execution of instructions in multiple threads (i.e., TLP) is possible and the system can seamlessly transition at run-time to the other models otherwise. In order to understand the tradeoffs involved, we also develop a performance model that allows one to quantitatively attribute overall performance gains to either TLP or ILP in such combined multithreaded execution model.

Experimental results show that our combined execution model achieves speedups of up to 41.2%, with an average of 10.2%, over an existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over a flavor of runahead execution for a subset of the SPEC2000 Integer benchmark suite.

We then investigate how a common ILP-enhancing microarchitectural feature, namely branch prediction, interacts with TLS. We show that branch prediction for TLS is even more important than it is for single core machines. Unfortunately, branch prediction for TLS systems is also inherently harder. Code partitioning and re-executions of squashed threads pollute the branch history making it harder for predictors to be accurate.

We thus propose to augment the hardware, so as to accommodate *Multi-Path* (MP) execution within the existing TLS protocol. Under the MP execution model, all paths following a number of *hard-to-predict* conditional branches are followed. MP execution thus, removes branches that would have been otherwise mispredicted helping in this way the processor to exploit more ILP. We show that with only minimal hardware support, one can combine these two execution models into a unified one, which can achieve far better performance than both TLS and MP execution.

Experimental results show that our combied execution model achieves speedups of up to 20.1%, with an average of 8.8%, over an existing state-of-the-art TLS system and speedups of up to 125%, with an average of 29.0%, when compared with multi-path execution for a subset of the SPEC2000 Integer benchmark suite.

Finally, Since systems that support speculative multithreading usually treat all threads equally, they are energy-inefficient. This inefficiency stems from the fact that speculation occasionally fails and, thus, power is spent on threads that will have to be discarded. We propose a profitability-based power allocation scheme, where we "steal" power from non-profitable threads and use it to speed up more useful ones. We evaluate our techniques for a state-of-the-art TLS system and show that, with minimal

hardware support, we achieve improvements in ED of up to 25.5% with an average of 18.9%, for a subset of the SPEC 2000 Integer benchmark suite.

Acknowledgements

First and foremost, I would like to thank my adviser Marcelo Cintra, for teaching me everything I know about Speculative Multithreading and for being supportive even when things were not going very well. Marcelo has been a model adviser and his way of thinking shaped me both as a researcher and as a person. I would like to thank Gregory Steffan, Mikel Luján and Björn Franke, for serving as my committee and for making my defense a nice experience. Their many useful suggestions, greatly enhanced this document. Many thanks to Stefanos Kaxiras who has been my mentor and the main reason I started working in computer architecture. In fact he was the one that introduced me to research and taught me how to think as a computer architect. Yanos Sazeides taught me many things about branch prediction and was always there to discuss my questions regarding process migration. I would like to thank Erik Hagersten for having me as an intern in ACUMEM. It was both a great experience and pleasure to work with him.

Although great advisers and mentors are important for completing successfully a PhD, colleagues and friends are equally important. I was lucky enough to be part of a large group that kept growing thanks to Mike O'Boyle, Marcelo Cintra, Björn Franke and Nigel Topham. I was fortunate enough to share an office with Salman Khan and Nikolas Ioannou, with which I also had the pleasure to work together throughout my years in Edinburgh. I would like to thank Alberto Ros Bardisa, Hugh Leather, Aristeides Efthymiou, John Cavazos, Grigori Fursin, Pedro Diaz, Zheng Wang, Damon Fenacci, Luís Góes, Christophe Dubach and Marcela Zuluaga for our interesting conversations regarding research, life and cooking! Sofia Pediaditaki was always a person to complain to and have coffee with, thanks for putting up with me. Georgios Tournavitis, has been a great friend, colleague and flatmate. Georgios has been my best friend throughout my stay in Edinburgh and has changed my way of thinking in many aspects.

On a more personal note, I would like to thank my friends in Greece Orestis Argiropoulos, Angelos Bastakis, George Kasapas, Georgios Keramidas, Pavlos Koutsoukos, Tasos Nanakos and Pavlos Petoumenos for always being there for me. Without them, going to Greece would have been only to visit my family. Kudos to Maria

Gaki who has been an important person for me since I met her, and came all the way to Edinburgh to visit me when I needed her the most. I would also like to thank Petros Dumont du Voitel and Caterina Stratoyianni for our long conversations over the phone and for supporting me. Thanks to Danae Malli for traveling all the way to Edinburgh to see me. Thanks to Nathalie Dumont du Voitel for the time we spent together. I always enjoyed my conversations with Magda and Kelly Houdi and our fun nights out. Andreas Tsiamis, Isaac Gual, Anja Riese, Alex Neumayr, Lola Vera Marco, Charis Efthymiou, Sayak Mitra, and Joanna Todd, made my life in Edinburgh much nicer, many thanks to all of you. I am lucky enough to have a small but very bonded family. Thanks to my grandparents Polychronis, Giannis, Mary and Evangelia and my uncles Simos and Stelios for spoiling me every time I saw them. My two sisters Evi and Mary Xekalaki have always been there for me. I am grateful for their love and affection. Finally, I would like to thank my parents Kostas and Tasoula for always believing in me (even when there was no clear reason to do so) and for always supporting me. Without them nothing would have ever been possible. Thank you for raising me up in the best possible way and for teaching me what is important in life and what is not.

Declaration

I declare that this thesis was composed by myself, that the work ontained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- “Combining Thread Level Speculation, Helper Threads, and Runahead Execution.”
  Polychronis Xekalakis, Nikolas Ioannou and Marcelo Cintra.
  International Conference on Supercomputing 2009.

- “Handling Branches in TLS Systems with Multi-Path Execution.”
  Polychronis Xekalakis and Marcelo Cintra.
  Review Pending.

- “Profitability-Based Power Allocation for Speculative Multithreaded Systems.”
  Polychronis Xekalakis, Nikolas Ioannou and Marcelo Cintra.
  Review Pending.

(Polychronis Xekalakis)

Contents

List of Tables

List of Figures

Chapter 1

Introduction, Contributions and Structure

1.1 Multi-Core Systems and Parallel Applications

With the shrinking of transistors continuing to follow Moore's Law and the non-scalability of conventional out-of-order processors, Multi-Core systems are becoming the design choice for both industry and academia. Performance extraction is thus largely alleviated from the hardware and placed on the programmer/compiler camp, who now have to expose Thread Level Parallelism (TLP) to the underlying system in the form of explicitly parallel applications.

Unfortunately, parallel programming is hard and error-prone. The programmer has to parallelize the work, perform the data placement, deal with thread synchronization and, of course, debug the applications (the non-determinism induced by the different noise on different cores makes this particularly hard). Although there has been extensive work on making synchronization easier [26, 35, 46], and on allowing deterministic replays for debugging purposes [37, 56], devising parallel algorithms and dealing with data placement is still cumbersome.

A convenient alternative to parallel programming is offered by parallelizing compilers [3, 9, 13, 47]. Parallelizing compilers are given sequential applications, which they try to parallelize by deducing that specific program segments do not contain any

data dependences. Despite many years of research, parallelizing compilers still fail to parallelize all but the most trivial applications. In fact as pointed out in [75] Intel's state-of-the-art compiler (ICC) actually gets a slowdown for most of the irregular applications it tries to parallelize. The main reason for this is that either they cannot *statically* guarantee that specific code segments do not have any dependences (e.g., if there is use of pointers) or they cannot find large enough sections where this holds so that their coverage is large.

1.2 Main Contributions

1.2.1 Combining TLS, HT, and RA Execution

As discussed above, with the advent of multi-core systems, the design effort has been alleviated from the hardware and placed instead on the compiler/programmer camp. Unfortunately parallel programming is hard and error-prone, sequential programming is still prevalent, and compilers still fail to automatically parallelize all but the most regular programs.

One possible solution to this problem is provided by systems that support Thread-Level Speculation (TLS) [32, 45, 53, 68, 71]. In these systems, the compiler/programmer is free to generate threads without having to consider all possible cross-thread data dependences. Parallel execution of threads then proceeds speculatively and the TLS system guarantees the original sequential semantics of the program. Thus, the TLS model improves overall performance by exploiting Thread-Level Parallelism (TLP) via the concurrent execution of instructions in multiple threads [1] (Figure 1.1(a)).

Another possible solution to accelerate program execution in multicore systems without resorting to parallel programs is provided by systems that support Helper Threads (HT) [15, 22, 73, 85]. In these systems, the compiler/programmer extracts small threads, often called slices, from the main thread such that their execution in parallel with the main thread will lead to improved execution efficiency of the latter. Most commonly, HTs are used to resolve highly unpredictable branches and cache misses

[1] In reality, the basic TLS execution model also provides some indirect non-TLP performance benefits as explained later in the thesis.

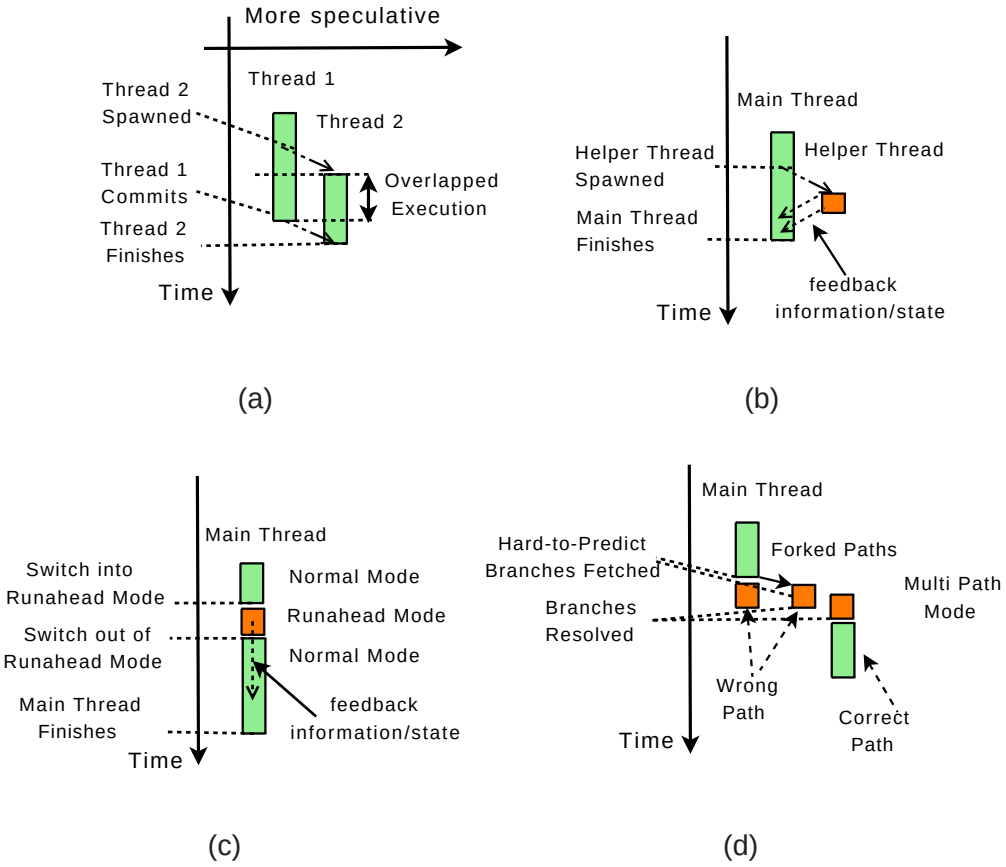

Figure 1.1: Different models of multithreaded execution: (a) Thread Level Speculation. (b) Helper Thread. (c) Runahead Execution. (d) Multi-Path Execution.

before these are required by the main thread. In almost all cases, the benefits of HT are indirect in the sense that no actual program computation is executed in parallel with the main thread. Thus, the HT model improves overall performance by exploiting Instruction-Level Parallelism (ILP) via the improved execution efficiency within a single main thread (Figure 1.1(b)).

A solution related to HT is to support Runahead (RA) execution [6, 14, 17, 28, 43, 58]. In these systems, the hardware transparently continues execution of instructions from the main thread past a long latency operation, such as a cache miss. Unlike with HT, the instructions in the runahead thread are not explicitly extracted and placed in a subordinate thread and are not executed concurrently with the main thread. In

fact, RA can be even implemented in single-core systems with some enhanced context checkpointing support. Like HT, the benefits from RA are indirect and it improves overall performance by exploiting ILP (Figure 1.1(c)).

Each of these three multithreaded execution models has been separately shown to improve overall performance of some sequential applications. However, given the different nature of the performance benefits provided by each model, one would expect that combining them in a combined execution model would lead to greater performance gains and over a wider range of applications compared to each model alone. Moreover, much of the architectural support required by each model is similar, namely: some support for checkpointing and/or multiple contexts, some support to maintain speculative (unsafe) data, and some support for squashing threads. Despite these opportunities no one (to the best of our knowledge) has attempted to combine these multithreaded execution models.

One contribution of this thesis is to combine these three multithreaded execution models in a super-set combined model that can exploit the benefits of each model depending on application characteristics. More specifically, the resulting system attempts to exploit TLP speculatively with TLS execution, but when this fails or when additional opportunities exist for exploiting ILP the system also employs a version of HT that is based on RA. We chose this model of HT so that its threads interact seamlessly with TLS threads and only small modifications to the TLS protocol and the TLS architectural support are required. In the thesis we discuss in detail this interaction and how to tune the HT and TLS models to work synergystically. Another contribution of this thesis is a simple methodology that allows one to model the performance gains with TLS and the combined execution model such that gains can be accurately attributed to either TLP or ILP. This methodology, then, allows one to reason about the behavior of the execution models and to investigate tradeoffs in the combined model.

Experimental results show that the combined execution model achieves speedups of up to 41.2%, with an average of 10.2%, over an existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over a flavor of RA execution for a subset of the SPEC2000 Int benchmark suite.

This component of the thesis is organized as follows. Section 3.4 presents our

methodology for quantifying the contributions to performance gains that come from TLP and ILP. Chapter 2 provides a brief description of the TLS, HT and RA execution models. Chapter 4 presents our proposed scheme to combine the three execution models. Chapter 5 presents results. Finally, Section 11.2 discusses related work.

1.2.2 Combining TLS and MP Execution

Another possible solution to accelerate program execution without resorting to parallel programs is provided by systems that support Multi Path (MP) Execution [2, 34, 44]. In these systems, the hardware fetches along two paths for the hard-to-predict branches (Figure 1.1(d)). The benefits of MP come from the fact that the processor is relieved from the misprediction penalty for the branches where it is applied. Some benefits can also come due to prefetching effects but typically they are not the main source of performance improvement over sequential execution. Thus, the MP model improves overall performance by exploiting ILP via the improved execution efficiency within a single main thread.

The vast majority of prior work on TLS systems has focused on architectural features directly related to the TLS support, such as protocols for multi-versioned caches and data dependence violation detection. More recently, it has been noticed that architectural features not directly related to the TLS support also have an important impact on the performance of the TLS system. Moreover, the performance impact of such features is sometimes other than intuition would expect given their well-understood impact on non-TLS systems. For instance, [77] shows that TLS systems benefit from a larger number of Miss Handling Resisters (MSHRs) than non-TLS systems can possibly exploit. Also, it is sometimes the case that better variations of such common architectural features can be found that are specifically tailored to TLS systems. For instance, [20] shows that re-using the history register from previous threads can boost branch prediction accuracy for TLS systems.

This part of the thesis focuses on the problem of branch prediction for TLS systems. Its contributions are as follows: We perform an in-depth study of the interaction between branch prediction and TLS performance. In this process, we show that the relationship of the performance of TLS systems to branch prediction is often hard to

model and sometimes different from that of the sequential execution. We then shed some light on the characteristics of TLS execution that impact branch prediction behavior. Based on the above observations, we propose to combine *MP Execution* with TLS as a means of resolving many hard-to-predict branches.

Experimental results show that the combined execution model achieves speedups of up to 20.1%, with an average of 8.8%, over an existing state-of-the-art TLS system and speedups of up to 125%, with an average of 29%, when compared with Multi-Path execution for a subset of the SPEC2000 Int benchmark suite.

This component of the thesis is organized as follows. Chapter 6 explores how branch prediction is affected by TLS and vice-versa, and motivates the combination of TLS with MP. Chapter 2 provides a brief description of the TLS and MP execution models. Chapter 7 presents our combined TLS and MP scheme which enhances TLS's performance by removing mispredictions of hard-to-predict branches. Chapter 8 presents results and Sections 11.5 and 11.6 discusses related work.

1.2.3 Profitability-Based Dynamic Power Allocation

Another contribution of this thesis is to identify threads that provide neither TLP, nor ILP benefits and distinguish them from those that do. By classifying threads into profitable and non-profitable ones, we can increase the efficiency of our speculative system by allocating power according to their profitability. More specifically, threads predicted to be non-profitable are put in one of the low power modes, allowing us to spend the power saved to accelerate the profitable ones. We guide our scheme based on two predictors: a dependence predictor able to detect lack of TLP, and a memory boundness predictor able to estimate lack of ILP.

Apart from being able to classify threads into profitable and non-profitable ones, we also require a mechanism to regulate the power resources accordingly. We could potentially implement this by supporting multiple types of cores (i.e., low power ones, high power ones etc.) and migrate the threads accordingly. However since our threads are by construction small (so as to require minimal hardware extensions in order to be able to buffer all the intermediate results), migration has serious performance repercussions. We instead choose to implement different power modes by performing Dynamic

Voltage and Frequency Scaling (DVFS) [52] on each core, which, when done by on-chip regulators, is a proven and fast way to trade power for performance [41].

Applying our profitability-based power allocation scheme to a state-of-the-art TLS system, we are able to achieve significant speedups with a reasonable increase in the power consumed. More specifically, by evaluating our technique for a subset of the SPEC2000 Int benchmarks, we show that with only minimal hardware support, we are able to achieve improvements in the overall Energy-Delay [2] (ED) of up to 25.5% with an average of 18.9%. Although the techniques proposed in this part of the thesis are evaluated only for a TLS system, they are directly applicable to a TM one.

This component of the thesis is organized as follows: Chapter 9 provides some background on DVFS and outlines our proposed scheme. Chapter 10 presents our results. Finally, Section 11.7 discusses related work.

1.3 Thesis Structure

Chapter 2 provides background information on the execution models that we propose to combine. Chapter 3 describes our experimental methodology. We provide details on the simulation infrastructure and the simulated baseline architecture. In subsequent chapters where we present the proposed schemes, we will iterate again over the baseline architecture, for a quick reference, and outline the additional hardware that is required. In the same chapter we present the compilation infrastructure used and we provide a brief description and analysis of the benchmarks used. We also present a statistical analysis using the Placket/Burman [60] technique that pinpoints the architectural bottlenecks for our baseline system. Finally we present a model that we will use in later sections to quantify the achieved speedups in terms of the respective ILP and TLP contributions.

Chapter 4 presents the proposed scheme and discusses implementation issues. In the same chapter we also discuss the extra hardware support required. Chapter 5 presents our experimental results and uses the model described earlier to analyze the source of the achieved speedups.

[2] Energy Delay is a combined metric, that is the product of the energy a system expended to perform an operation with the time it required to perform it.

Chapter 6 quantifies what the impact of branch prediction is on TLS and shows that it is more important than it is for sequential systems. The same chapter also shows, using the entropy of branches, that branch prediction is harder for TLS and that high-end branch predictors are likely to provide only minimal improvements. Chapter 7 describes how one can lessen the branch prediction problem by allowing speculative threads to follow multiple paths. Chapter 8 presents experimental results that quantify the performance of the proposed scheme and also performs a sensitivity analysis of the design parameters.

In Section 1.2.3 we argued for a profitability-based power allocation scheme for TLS systems, acknowledging the fact that speculating after a point comes at diminishing returns. In fact even for the profiled base TLS systems, some tasks are more energy efficient than others. Chapter 9 describes the proposed scheme and how we can design profitability predictors to achieve runtime adaptation. Chapter 10 presents experimental results that clearly show the benefits of the proposed scheme.

Chapter 11 discusses related work and Chapter 12 summarizes the contributions of this thesis. Chapter 13 describes possible future work based on the proposed schemes and provides a few concluding remarks.

Chapter 2

Background on TLS, HT, RA and MP Execution Models

2.1 TLS

Under the *thread-level speculation* (also called *speculative parallelization* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics [32, 45, 53, 68, 71]. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores from speculative threads generate unsafe *versions* of variables that are stored in some sort of *speculative buffer*. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads from speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. Any value read from a predecessor thread, is called an *exposed read*, and it has to be tracked since it may expose a Read-After-Write (RAW) dependence. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be re-executed. When the execution of a non-speculative thread completes it *commits*

and the values it generated can be moved to safe storage (usually main memory or some shared higher-level cache). At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the processor is free to start executing a new speculative thread. An example of TLS execution is depicted in Figure 2.1, where a loop is speculatively parallelized. While there are no dependences among the threads created from iterations *j* and *j+1*, this is not the case for the one created from iteration *j+2*. In fact this thread has a RAW dependence with thread *j* and as such it has to be restarted and its data discarded.

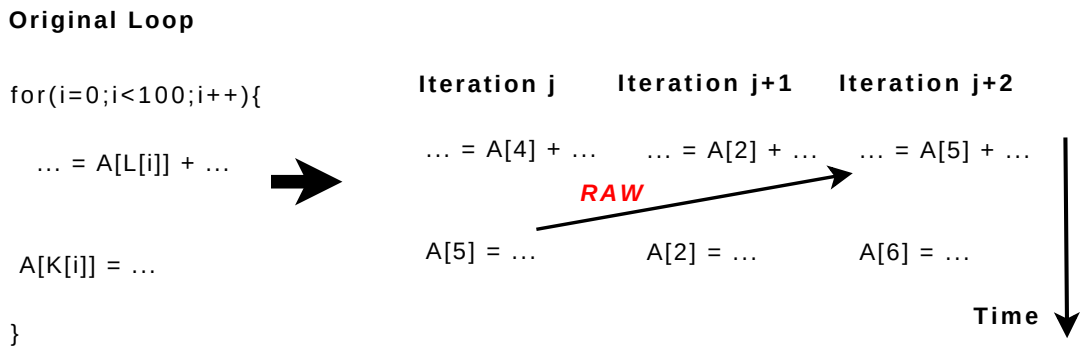

Figure 2.1: Speculatively parallelizing a loop with TLS. Iterations of a loop are converted to threads. Iteration j+2 violates the sequential semantics (RAW dependence) and as such it has to be squashed.

Speculative threads are usually extracted from either loop iterations or function continuations. The compiler marks these structures with a fork-like *spawn instruction*, so that the execution of such an instruction leads to a new speculative thread. The *parent* thread continues execution as normal, while the *child* thread is mapped to any available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread (Figure 2.2(a)). Threads formed from iterations of the same loop (and that, thus, have the same spawn point) are called *sibling* threads. For function calls, spawn points are placed just before the function call, so that the non-speculative thread proceeds to the body of the function and a speculative thread is created from the function's

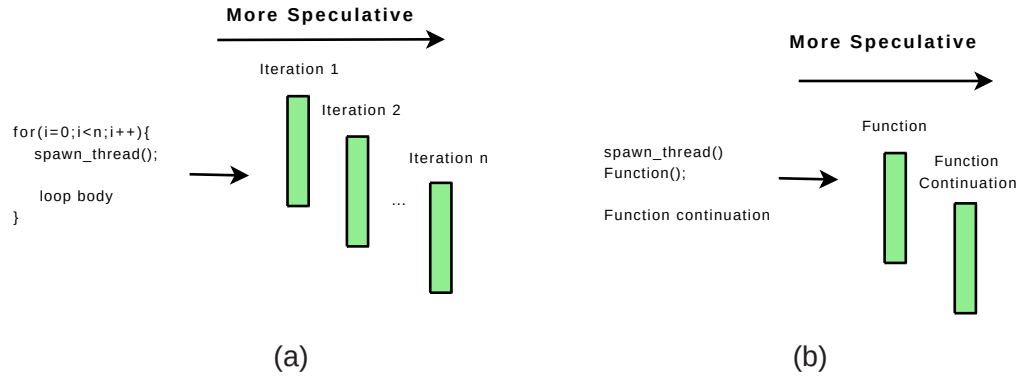

Figure 2.2: Extracting threads from: (a) Loop iterations. (b) Function continuations.

continuation (Figure 2.2(b)). In order to extract more parallelism, it is also possible to create speculative threads from nested subroutines and loop iterations. Under these schemes threads are spawned in strict reverse order, more speculative first, compared to their sequential execution. Such schemes are said to have *out-of-order spawn* and have been shown to provide significant performance benefits [63].

2.1.1 Architectural Support

In this section we describe the hardware support that is required so as to fully support TLS on an existing multi-core system. The architectural support required for any TLS system consists of six components: i) a mechanism to allow speculative threads to operate in their own context and to enforce that speculatively modified data be also kept separate, ii) a mechanism to track data accesses in order to detect any data dependence violations, iii) a mechanism to spawn threads in different cores or contexts, iv) a mechanism to rollback (i.e., squash and restart) incorrectly executed threads, v) a mechanism to commit the correctly speculatively modified data to the safe state, and vi) a mechanism to keep track of the ordering of threads with respect to the original sequential execution order. Although there are many possible ways to support these mechanisms, throughout this thesis we assume as our baseline architecture the one proposed in [63], which has been shown to excel in both performance and energy efficiency. Details about each of these operations are given in the following sections.

2.1.1.1 Data Versioning

Memory accesses issued by speculative tasks must be carefully handled so as not to compromise the correct execution of the application. In particular, since some of the speculative tasks may be incorrect, their state cannot be merged with that of the safe state of the program. Consequently, the state of each thread is stored separately, typically in the cache of the processor running the task. Additionally, stores are not allowed to propagate to lower memory levels. 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.

Each task has at most a single version of any given variable. In order to maintain the correct execution semantics, we have to enforce that each task buffers its intermediate values. Since some of the tasks may have to perform stores to the same variables, and thus produce different values for the same variables, we must buffer them separately. Additionally, reads performed by speculative tasks have to be performed based on the ordering they would have had, if they were executed sequentially. This can be achieved if each task read is provided with the closest predecessor version of the variable. Finally, the variables that have been updated during speculative execution have to be committed based on the ordering of the tasks. Typically, all three operations leverage on a special type of cache, the *Speculative Versioning Cache* (SVC) [30].

Such a multi-versioned cache can hold state from multiple tasks by tagging each cache line with a version ID, which records what task the line belongs to. Intuitively, such version ID could be the task version. In addition to facilitating the data versioning and data movement, there are also two performance reasons why multi-versioned caches are desirable: they avoid processor stalls when tasks are imbalanced, and enable lazy commit. If tasks have load imbalance, a processor may finish a task and the task is still 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 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. Lazy commit is an approach where, when a task commits, it does not eagerly merge its cache state with main memory through ownership requests [70] or write backs [45]. 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.

2.1.1.2 Detecting 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 either word or cache line level. Although tracking dependences at the word granularity minimizes the amount of false dependences (caused when threads write a different byte from the one read), the additional hardware cost of doing so renders it a less attractive approach than tracking dependences at the cache line level. In all 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.

When a read is performed, the write bit is checked. If this bit is not set, then the value is read from a predecessor task and as such it is an exposed read. Exposed reads are marked by setting a bit for the specific cache line (or word depending on the granularity). A data dependence violation occurs when a task writes a location that has been read by a successor task with an exposed read. In order to unveil these violations, the addresses of the performed writes have to appear on the bus, so that the remaining cores can snoop the bus and perform a check of whether they have performed an exposed read for that address.

In addition to data dependence violations, tasks are also exposed to control dependence violations. Control violations occur when a task is spawned in a mispredicted branch path.

2.1.1.3 Spawning Threads

Spawning a new thread is a process that in conventional architectures is typically fairly slow. In TLS systems, where thread spawns are fairly frequent, this would impair performance. For this reason special support for fast spawning of threads is required. More specifically, in TLS systems when a thread encounters a thread spawn instruction, it creates a small packet containing the stack pointer, the program counter and some counters that have to do with the thread ordering. This packet is sent to an empty core which can start execution immediately after initializing its program counter and stack pointer accordingly. Instead of relying on register communication, as Multiscalar [68] does, under our framework communication of live-ins is done through memory. The compiler ensures that all values that are live-ins for the newly created thread will be spilled into memory, so that when the new thread requests them they will be propagated to it via the TLS protocol.

2.1.1.4 Rolling Back

Rolling back any changes is a fairly important architectural component of TLS systems. TLS threads should be able to restore any changes, so that the architectural state remains valid even when data dependence violations have occurred. When a violation is detected (control or value), the pipeline and the store buffers are flushed. The cache lines in the speculative buffer that are not dirty and have not been modified by any other thread, are kept intact whereas the rest of the cache lines are invalidated. Squashes come in two forms. In a control violation, the task is squashed with a 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 another data dependence. If the thread is restarted or killed, the register state is discarded. For a restart the stack pointer and program counter are reset to their initial values.

2.1.1.5 Committing State

Committing state can only happen when a thread becomes non-speculative. Becoming non-speculative implies that the thread can no longer violate any of the sequential

semantics and as such cannot perform any operations on incorrect data. When such a thread finishes execution, any cache lines it modified have to be written back to memory. This is typically done via a lazy policy, where lines that should be written back are left in the caches until they are replaced and thus written back to a lower level cache. With multi-versioned caches, execution of subsequent threads can proceed despite this effect.

2.1.1.6 Maintaining Thread Ordering

Thread ordering ensures that when threads commit, they commit in an order imposed by the sequential semantics. For systems that only support in-order spawning of threads, maintaining thread ordering is straightforward - it is done by maintaining a global counter which is incremented before it is passed to any of the children threads. Out-of-order spawning makes things much harder. Under out-of-order spawning, threads are spawned in strictly reverse order than their sequential semantics. Thread ordering here is maintained via splitting timestamps. [63] was the first to propose such a system, so that the correct ordering is maintained in a distributed fashion.

2.1.2 Compilation Support

Generating TLS binaries requires some modest compilation support so as to partition the program into tasks. Each 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 from 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.

Generally a TLS compiler consists 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. Tasks with little potential are eliminated by the task pruning pass. The live-ins are calculated for the remaining tasks and spill and re-load code is inserted before the spawn point and after the begin point, respectively.

Additional compilation support can help significantly, although it is not generally required for correct TLS execution. The compiler can remove some of the dependence violations either by performing software value prediction for the induction variables so as to remove some of the dependences that might cause violations or by explicitly synchronizing tasks. The POSH [49] compiler is an example of a compiler that uses value prediction. However, POSH does so only for what it is able to identify as induction variables. The Mitosis [61] compiler adopts a much more general method for value prediction, inserting precomputation slices at the start of speculative sections. This is done by traversing the control flow graph backwards starting at the begin point. Instructions that produce the live-ins to the speculative section are selected. These instructions are then duplicated at the start of the speculative section.

Although the great advantage of TLS is that synchronization between tasks can be avoided, often dependences causing squashes in speculatively executed loop iterations can be statically determined. This observation is exploited in [84] by introducing synchronization instructions when dependences between loop iterations can be identified at compile time. Mispeculation is avoided by inserting wait and signal primitives, forcing uses to wait until the producing instruction in the predecessor thread has executed.

2.2 HT

Under the *helper threads* (also called *subordinate microthreads*) approach [15, 22, 73, 85], small threads are run concurrently with a main thread. The purpose of the helper threads is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Common ways to accelerate the execution of the main thread involve initiating memory requests ahead of time (i.e., prefetching; such that the results are hopefully in the cache by the time they are needed by the main thread) and resolving branches ahead of time. An example of this is shown in Figure 2.4, where a loop suffers from a branch misprediction due to a hard-to-predict branch and subsequently also suffers a cache miss. By taking the backward slice leading to the cache miss instruction, a helper thread can be created, which when spawned ahead of time, is able

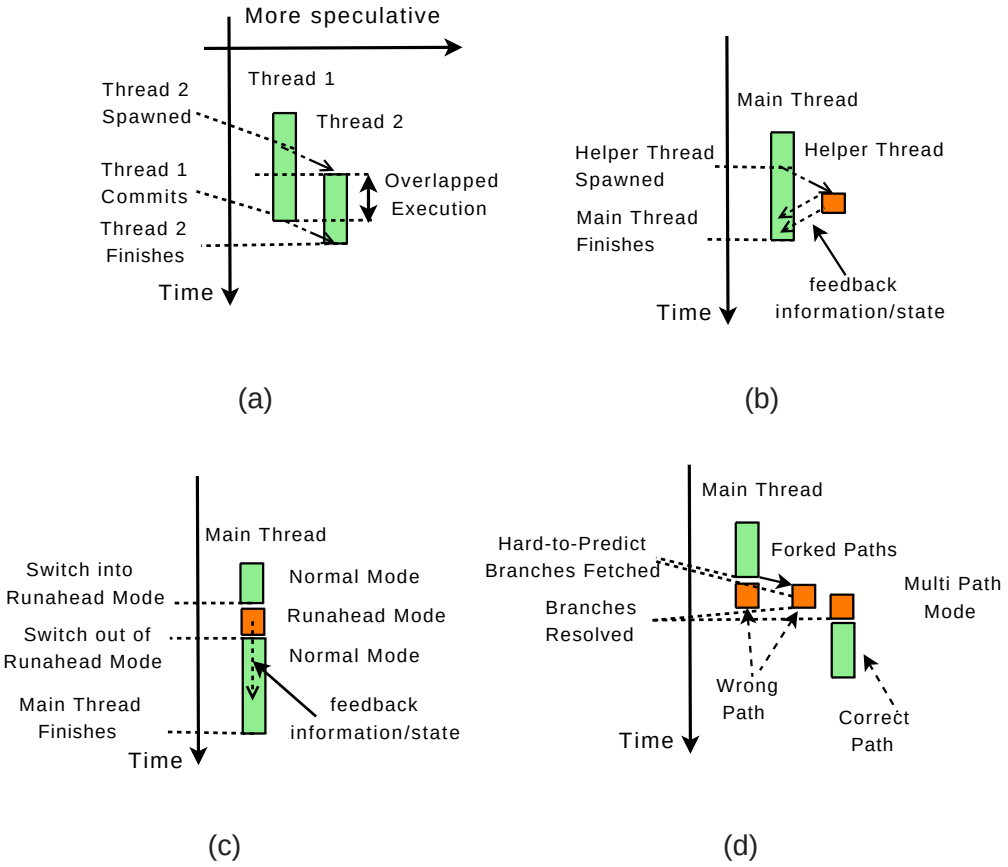

Figure 2.3: Different models of multithreaded execution: (a) Thread Level Speculation. (b) Helper Thread. (c) Runahead Execution. (d) Multi-Path Execution.

to prefetch/warm-up for the main thread and thus improve its ILP.

Usually, depending on how the helper threads are generated (see below), the execution of helper threads is speculative in that they may be following some incorrect control flow path and/or producing and consuming incorrect data. In this multithreaded execution model there is no particular ordering among multiple helper threads and all are *discarded* at the end of their execution. Figure 2.3(b) depicts this execution model.

Helper threads are usually generated by the compiler or programmer and often consist of *slices* of instructions from the main thread (e.g., only those instructions directly involved in the computation of some memory address or branch condition). Depending on the size and complexity of the helper threads it may be possible to

keep all their intermediate results in the registers, but it may be necessary to allow for spills to the memory hierarchy, which in turn requires providing storage for speculative *versions* of data. The compiler marks the main thread with fork-like *spawn instructions* at points where particular helper threads should be initiated.

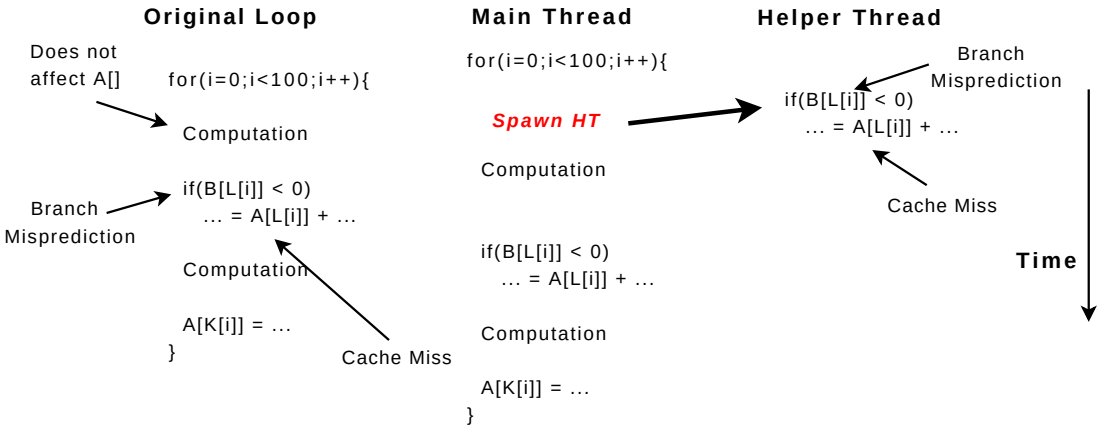

Figure 2.4: Helper threads can help with hard-to-predict branches and memory misses. The original loop suffers from a branch misprediction and a memory miss. The modified loop spawns a helper thread that is able to resolve the misprediction and the cache miss, so that when the main thread reaches that point it is able to predict the branch correctly and find the requested cache line.

The architectural support required by HT consists of three main components: i) a mechanism to allow helper threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to spawn threads in different cores or contexts, and iii) a mechanism to discard threads when finished.

2.3 RA

Under the *Runahead* approach [6, 14, 17, 28, 43, 58], when the main thread hits a long-latency operation (e.g., an L2 miss) it halts execution and a runahead thread continues execution either ignoring or predicting the outcome of the long-latency operation. The

purpose of the runahead thread is not to directly contribute to the actual program computation, which is often still performed in full by the main thread once it resumes, but to facilitate the execution of the main thread indirectly after it resumes. As with HT, common ways to accelerate the execution of the main thread involve prefetching and early branch resolution. Unlike HT, runahead threads do not run concurrently with the main thread. Thus runahead execution can be seen as a form of HTs which run only on idle cycles of the processor. The execution of the runahead thread is speculative since the outcome of the long-latency operation is either ignored or predicted. Thus, in most proposed models the runahead thread is *discarded* once the main thread resumes execution. In more aggressive models, however, if the predicted outcome of the long-latency operation is correct the execution of the runahead thread is incorporated into the main thread before stopping the execution of the runahead thread. Figure 2.3(c) depicts this execution model.

Runahead threads are generated on-the-fly by the hardware and, like the common HT case, consist of a selection of instructions from the main thread. Strictly speaking, in many proposals in the literature, the runahead threads are in fact obtained by simply *checkpointing* the main thread and letting it run ahead instead of explicitly spawning a new thread elsewhere. Also like HT, it may be possible to keep all the intermediate results of the runahead thread in the registers, but it may be necessary to allow for spills to the memory hierarchy.

The architectural support required by RA consists of five main components: i) a mechanism to allow runahead threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to decide when to generate runahead threads or to switch the main thread into runahead mode, iii) a mechanism to discard incorrectly executed threads, and iv) and v) optional mechanisms to check if the runahead thread has executed based on correct or incorrect predicted outcomes and, if so, to incorporate the runahead state and data into the main thread.

2.4 MP

Under the *Multi-Path* (MP) execution model, both paths following a number of *hard-to-predict* conditional branches are followed. More specifically when a branch is thought [1] to be a hard-to-predict one, another thread is created which is given a copy of the register file as it is before executing the branch. In order to allow fast register file copying, all threads are usually executed on the same core, which has multi-threading support. When the branch that triggered the MP execution is resolved, the thread that lies on the wrong path is discarded. Threads in MP mode cannot commit their state, since they might be executing instructions on the wrong path, thus intermediate stores are not propagated to the cache hierarchy - they are instead accommodated in the store buffers (in this model no spills are allowed). While executing in MP mode if there is no context available, subsequent hard-to-predict conditional branches are typically treated as normal branches, that is, they are predicted using the branch predictor. MP is thus able to avoid branch mispredictions at the cost of executing more instructions.

The architectural support required by MP consists of three main components: i) a mechanism to allow MP threads to operate in their own context and, possibly, to enforce that speculatively modified data be also kept separate, ii) a mechanism to decide when to generate MP threads, and iii) a mechanism to discard incorrectly executed threads.

Table 2.1 summarizes the architectural support required by the four multithreaded execution models in columns 2 to 5 (the last column shows the support used by our combined TLS/HT/RA scheme, which is described in Section 4).

[1] Confidence estimators are typically used to predict whether the branch predictor usually fails to predict correctly the specific branch.

Table 2.1: Hardware support required for the different models of multithreaded execution and for our combined approach. *O* stands for optional feature, *X* stands for feature not required, and ✓stands for feature required.

Mechanism	**TLS**	**HT**	**RA**	**MP**	**Combined**
Data Versioning	✓	✓	✓	✓	✓
Data Dep. Tracking	✓	X	X	X	✓
Spawn Threads	✓	✓	X	✓	✓
Discard/Rollback	✓	✓	✓	✓	✓
Commit State	✓	X	O	X	✓
Order Threads	✓	X	X	X	✓
Checkpoint Threads	O	X	✓	X	✓
Value Predict L2 Misses	X	X	O	X	✓

Chapter 3

Evaluation Methodology

Evaluating any proposal in computer architecture, as with any other science, requires experimental validation. This can be done either by implementing and fabricating a new chip, that will implement the proposed idea or by simulating it. Although there are many advantages to fabricating a chip, its cost is prohibitive in most cases, and as such simulation is the tool of choice for the architectural community.

Simulators are programs that mimic what a real processor would do when running a specific application. Because simulators are programs, it is easy to instrument any event we wish and it is relatively easy to implement oracle schemes so as to perform limit studies. Typically, simulators have errors when compared to real processors that are in the order of 10%, however since the comparisons between two schemes (the base case processor and the newly proposed one) use the same simulator, results are relative and as such this error may not be as important. Throughout this study we have used the SESC simulator [64], which has been shown to be quite accurate when compared with a MIPS R1000 (less than 4% error). We provide more details regarding our experimental methodology in the following sections.

3.1 Simulation Environment

We conduct our experiments using the SESC simulator [64]. In SESC, the actual instructions are executed in an emulation module, which emulates the MIPS Instruction

Set Architecture (ISA). It emulates the instructions in the application binary in order. The emulation module is built from MINT, a MIPS emulator. The emulator returns instruction objects to SESC which are then used for the timing simulator. These instruction objects contain all the relevant information necessary for accurate timing. This includes the address of the instruction, the addresses of any loads or stores to memory, the source and destination registers, and the functional units used by the instruction. The bulk of the simulator uses this information to calculate how much time it takes for the instruction to execute through the pipeline.

The main microarchitectural features are listed in Table 3.1. The system we simulate is a multicore with 4 processors, where each processor is 4-issue out-of-order superscalar. The branch predictor is a hybrid bimodal-gshare predictor. The minimum branch misprediction latency is 12 cycles while we also employ speculative updates of the global history register along the lines of [40]. Each processor has a multi-versioned L1 data cache and a non-versioned L1 instruction cache. All processors share a non-versioned unified L2 cache. For the TLS protocol we assume out-of-order spawning [63]. The latencies of all the caches were computed based on CACTI [74] for a 70nm technology. The power consumption numbers are extracted using CACTI [74] and wattch [11].

3.2 Compilation Environment

The TLS binaries were obtained with the POSH infrastructure [49]. For reference, the sequential (non-TLS) binaries where obtained with unmodified code compiled with the MIPSPro SGI compiler at the O3 optimization level. In order to directly compare them, we need to make sure that both the sequential and the TLS system, execute the same code segments. Traditionally, this is ensured by executing a given number of instructions. For TLS systems however, counting the number of instructions does not guarantee anything, since we speculatively execute many more instructions. For this reason, we place simulation marks across the code regions we wish to simulate and make sure that both the sequential and the TLS systems, execute the code segment between them. This is also necessary because the binaries are different, due to

Table 3.1: Baseline architectural parameters.

Parameter	TLS (4 cores)
Frequency	4GHz
Fetch/Issue/Retire Width	4, 4, 4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80, 104
Ld/St Queue	54, 46
Branch Predictor	48Kbit Hybrid Bimodal-Gshare
BTB/RAS	2K entries, 2-way, 32 entries
Minimum Misprediction	12 cycles
Task Containers per Core	8
Cycles to Spawn	20
Cycles from Violation to Kill/Restart	20

re-arrangements of the code by POSH. Note that these simulation marks have to be placed in locations where there is no speculation happening, otherwise threads that miss speculate might incorrectly end the simulation prematurely. We simulate enough simulation marks so that the corresponding sequential application graduates more than 750 million instructions, after skipping the initialization phase.

3.3 Benchmarks

We use the integer programs from the SPEC CPU 2000 benchmark suite [72] running the Reference data set. We use the entire suite except *eon*, which cannot be compiled because our infrastructure does not support C++, and *gcc* and *perlbmk*, which failed to compile in our infrastructure. In the next sections we briefly describe each one of the benchmarks and analyze them both for the sequential and the TLS case.

3.3.1 Description of Benchmarks

Bzip2: 256.bzip2 is based on Julian Seward's bzip2 version 0.1. The only difference between bzip2 0.1 and 256.bzip2 is that SPEC's version of bzip2 performs no file I/O other than reading the input. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.
Crafty: Crafty is a high-performance computer chess program that is designed around a 64bit word. It is primarily an integer code, with a significant number of logical operations such as *AND*, *OR*, *XOR* and *SHIFT*.
Gap: Gap implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
Gzip: Gzip (GNU zip) is a popular data compression program which is part of the GNU project. It uses Lempel-Ziv coding (LZ77) as its compression algorithm.
Mcf: A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic.
Parser: The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of a set of labeled links connecting pairs of words.
Twolf: The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells) which constitute the microchip.
Vortex: VORTEx is a single-user object-oriented database transaction benchmark which exercises a system kernel coded in C. The VORTEx benchmark is a derivative of a full OODBMS that has been customized to conform to SPEC CINT2000 (component measurement) guidelines.
Vpr: Vpr is a placement and routing program. It automatically implements a technology mapped circuit (i.e., a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip. It is an example of an integrated circuit computer-aided design program, and

algorithmically it belongs to the combinatorial optimization class of programs.

3.3.2 Brief Analysis of Benchmark Characteristics

3.3.2.1 Code Bloat

Figure 3.1 depicts the static code bloat introduced by the compiler (i.e., the standard POSH infrastructure). Code bloat results from the extra loads and stores the compiler has to insert in order to pass the live-ins on task creation. In addition to this the compiler inserts extra instructions to spawn new tasks and to mark their end. On average the TLS binaries have 11% more instructions than the sequential ones. Note that depending on the program, code bloat varies significantly and in some cases like *mcf* and *parser* it may be fairly large.

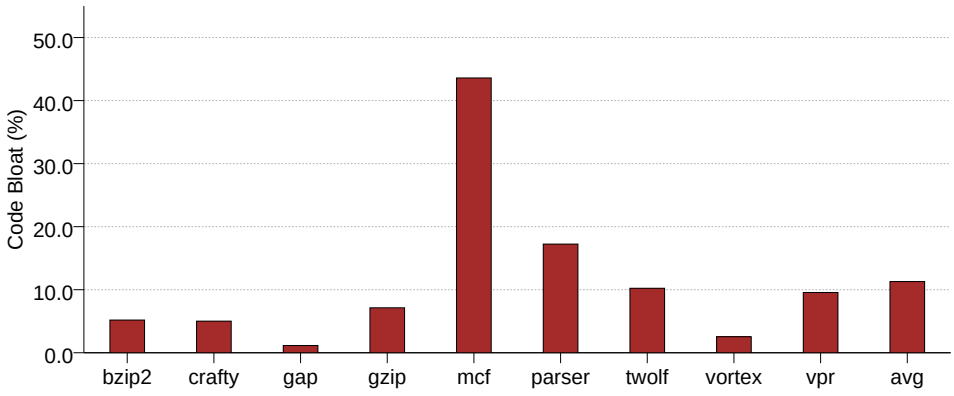

Figure 3.1: Code bloat per application.

3.3.2.2 Task Sizes

Partitioning applications to tasks that are small enough so that their speculative data can fit in the versioned caches, while being large enough to procure TLP benefits, is a crucial aspect of TLS systems. Figure 3.2 shows a breakdown of the task sizes for each of the benchmarks considered. On average more than 70% of the tasks are smaller than 500 instructions, while a reasonable number of threads with more than 2000 instructions exist in only three applications (i.e., *crafty, gap*, and *vortex*).

Note that these tasks are the ones that the POSH profiler found to be profitable. Larger tasks either had dependences that would not allow them to provide any TLP benefits, caused overflows in the speculative buffers, or suffered from load imbalance so that smaller tasks were favored. Unfortunately, small tasks render the overall performance fairly sensitive to ILP.

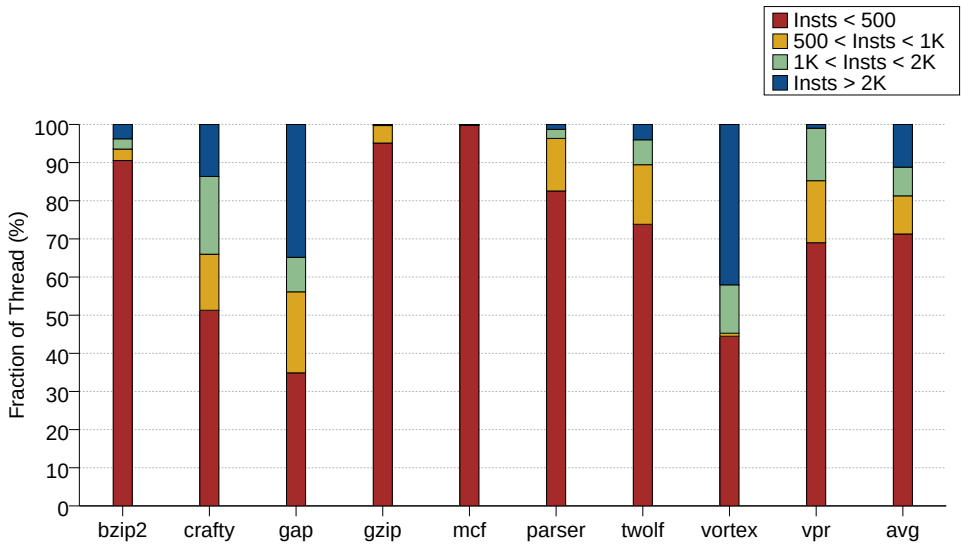

Figure 3.2: Fraction of threads of different sizes.

3.3.2.3 Dependence Violations

Even after profiling, some tasks violate the sequential semantics and as such they must be squashed. Figure 3.3 shows the fraction of cycles spent in tasks that squash over the total time spent executing all tasks. From all the applications, *gap* is the worst, spending 31.5% of its time executing tasks that will not provide any TLP benefits. *Twolf* and *bzip2* on the other hand have only a small number of violations and spend 4.0% and 7.4% of their time in such tasks, respectively.

Figure 3.4 shows how the number of squashes per core varies with the number of cores. Note that while the number of squashes per core increases when we move from two to four cores, this is not the case for some of the applications when we

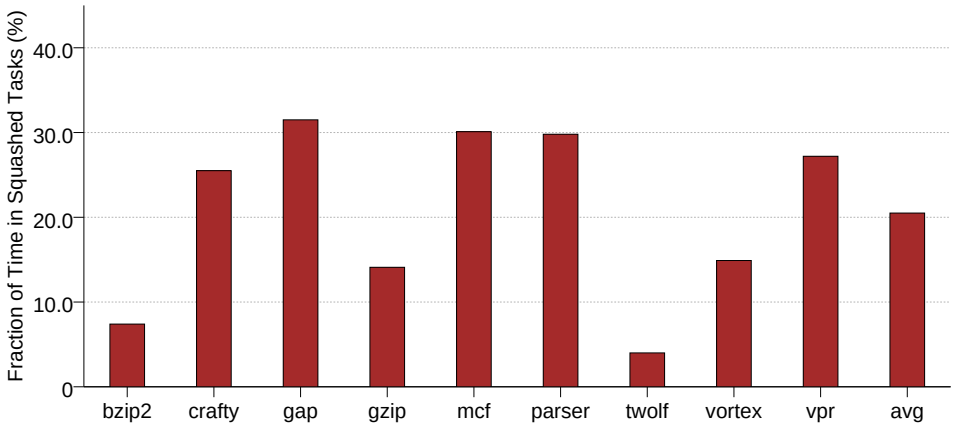

Figure 3.3: Fraction of time spent in tasks that squash over time spent for all tasks.

move from four to eight cores. The increase in the number of squashes per core in the former case is the result of the deeper speculation that is possible thanks to a larger number of cores (some of the thread spawns are suppressed for the two core case, leading to serialization). This in turn results in the creation of tasks that are fairly probable to either cause a violation themselves or have a predecessor thread that caused a violation. When we move to a larger number of cores, however, we are not always able to create enough threads so as to utilize all the cores. This brings the average number of squashes per core for the eight core system, down when compared with the four core case. Note however, that the total number of squashes increases almost linearly with the number of cores. This graph suggests that from a power perspective TLS is prohibitive for a large number of cores, since the amount of threads that do not provide benefits increases significantly.

3.3.3 Microarchitectural Bottlenecks of TLS Systems

Before trying to optimize cores for TLS, it is essential to understand and quantify the microarchitectural bottlenecks for TLS systems. Instead of relying on design experience, we advocate the use of a statistically rigorous scheme, able to quantify the relative importance of the microarchitectural features for the system's performance. More specifically, we use the Placket-Burman (PB) design pattern [60].

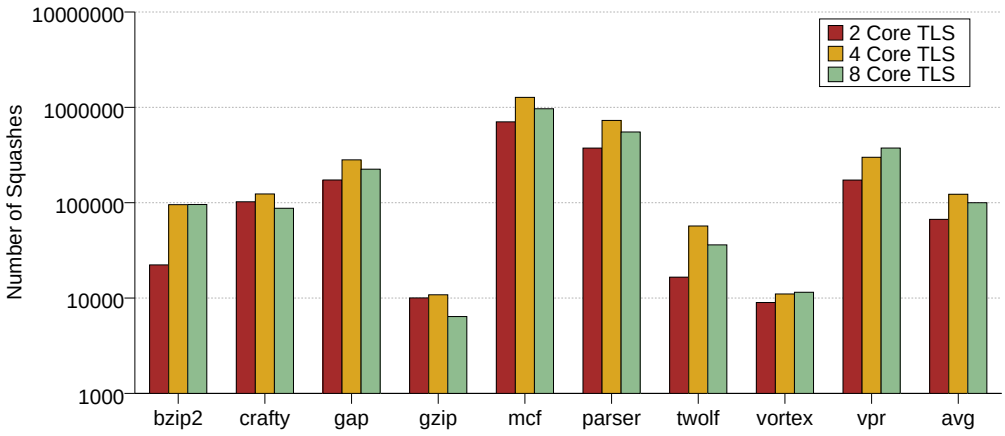

Figure 3.4: Average number of squashes per core with varying number of cores.

Under this design pattern, we first pick the design parameters of interest. We then define for each parameter two values: the one we expect our system to have and a larger than normal one. We then proceed to construct a design matrix. The rows of the design matrix correspond to different processor configurations while the columns correspond to the values of the parameters for each of the configurations. Since the Placket-Burman designs exist only for systems with parameters that are multiples of four, for the construction of the design matrix we need X simulations, where X is the next multiple of four that is greater than the number of parameters. When there are more columns than parameters, the extra columns serve as placeholders and have no effect on the simulation results. With the simulation results, we can then fill in the matrix with the performance (or power) for each of the configurations and from that compute the impact of each of the parameters on the end performance.

From all the possible design parameters, we picked the seven more important and we then performed the aforementioned analysis for both the sequential case and for a four core TLS system. More specifically, we picked as parameters the size of the level two cache, the size of the level one data cache, the size of the level one instruction cache, the size of the register file, the size of the ROB, the branch predictor and the size of the load/store queue. The *Normal* value for each of the parameters as well as the *High* and *Low* ones can be seen in Table 3.2 and correspond to the baseline val-

Table 3.2: Values of architectural parameters used for the Placket Burman exploration.

Parameter	Normal	High	Low
L2 Cache	1MB	2MB	512KB
L1 DCache	16KB	32KB	8KB
L1 ICache	16KB	32KB	8KB
Reg. File	80	160	40
ROB	104	208	52
Branch Predictor	48Kbit	96Kbit	24Kbit
Ld/St Queue	54/54	108/108	27/27

ues shown earlier in Table 3.1. We then performed simulations for the configurations shown on Table 3.3 and collected the execution times for each of them. In Table 3.3 processor configurations can be read by inspecting its rows, where a *-1* refers to a *Low* value and *+1* corresponds to a *High* one for the corresponding parameter.

Using this technique we found that the bottlenecks witnessed for the sequential execution were the same with that of TLS execution, although their effect was more pronounced in the latter. This suggests that microarchitectural features will limit the performance more for TLS system, than they would for simple sequential execution, and as such they should be more closely examined. The relative importance of the features can be found in Table 3.4. We see that a larger ROB generally helps significantly, as does a better memory system. We also see that branch prediction is in the top four bottlenecks. These numbers are in line with those presented in [83]. It is important to note that this ranking is done for the simulated regions of code, which where picked so as not to favor TLS over sequential execution (i.e., this could be done by simulating only the parallelizable loops) and as such we believe it is representative of what would be the bottlenecks of a real TLS system. The techniques proposed in later sections aim at improving TLS execution by targeting exactly these bottlenecks. More specifically, by combining TLS with HT and RA (Chapter 4) we are able to both improve on the

Table 3.3: Plackett and Burman design with foldover (X=8).

L2 Cache	L1 DCache	L1 ICache	Reg. File	ROB	BPred	LD/ST
+1	+1	+1	-1	+1	-1	-1
-1	+1	+1	+1	-1	+1	-1
-1	-1	+1	+1	+1	-1	+1
+1	-1	-1	+1	+1	+1	-1
-1	+1	-1	-1	+1	+1	+1
+1	-1	+1	-1	-1	+1	+1
+1	+1	-1	+1	-1	-1	+1
+1	+1	+1	-1	+1	-1	-1
-1	-1	-1	+1	-1	+1	+1
+1	-1	-1	-1	+1	-1	+1
+1	+1	-1	-1	-1	+1	-1
-1	+1	+1	-1	-1	-1	+1
+1	-1	+1	+1	-1	-1	-1
-1	+1	-1	+1	+1	-1	-1
-1	-1	+1	-1	+1	+1	-1
-1	-1	-1	+1	-1	+1	+1

Table 3.4: Ranking of importance of parameters according to the Placket-Burman technique (1-Better, 7-Worse). The ranking is the same for sequential and TLS executions.

Ranking	Parameter
1	ROB
2	L2cache
3	DCache
4	Bpred
5	Reg. File
6	LD/ST Queue
7	ICache

level two cache behavior and to increase clustering of cache misses (the effect a larger ROB would be similar). By combing TLS with MP execution (Chapter 7), we improve the branch prediction capabilities of the cores.

3.4 Quantifying Performance Gains in Speculative Multithreaded Executions

With multithreaded execution models part of the performance variation observed is due to overlapped execution of instructions from multiple threads where these instructions contribute to the overall computation – we call this a TLP contribution. Another part of the performance variation is due to indirect contributions that improve (or degrade) the efficiency of execution of the threads – we call this an ILP contribution. For instance, with TLS the parallel execution of threads leads to a TLP contribution but also prefetching effects may lead to an ILP contribution. This may happen when some threads share data so that the first thread to incur a cache miss effectively prefetches for the others such that the others will appear to have an improved ILP. Note that it is

also possible that due to contention for resources, threads appear to have a degraded ILP. Note also that with speculative multithreaded models the possibility of squashes and re-execution of threads leads to even more intricate relationships between TLP and ILP contributions. Accurately quantifying the TLP and ILP contributions toward the final observed performance variation is critical in order to reason and act upon the behavior of multithreaded execution models. In this section we present a methodology to quantify these TLP and ILP contributions in multithreaded execution models using easy to collect timing information from the actual execution. The model is a variation of that proposed in [63]: it requires one extra simulation run but provides a more accurate estimate of the ILP contribution (Section 3.4.1).

The performance model is based on measuring the following quantities from the execution: 1) the execution time of the original sequential code (T_{seq}); 2) the execution time of the modified TLS code when executed in a single core (T_{1p}); 3) the sum of execution times among all threads that actually *commit* ($\sum T_i$, for all threads i that commit); and 4) the execution time of the modified TLS code when executed in multiple cores (T_{mt}). Figure 3.5 depicts these quantities for a simple example with two threads. With these quantities, the overall performance variation (S_{all}) is given by Equation 3.1 and the performance variation (usually a slowdown) due to the TLS instrumentation overhead (S_{1p}) is given by Equation 3.2. The latter is needed in order to account for the variations needed in the binaries that execute the sequential and the multithreaded versions of the program.

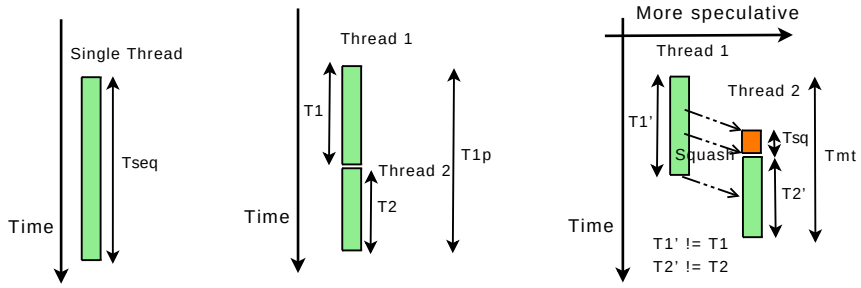

Figure 3.5: Quantifying ILP and TLP benefits.

$$S_{all} = \frac{T_{seq}}{T_{mt}} \tag{3.1}$$

$$S_{1p} = \frac{T_{seq}}{T_{1p}} \tag{3.2}$$

The overall performance variation of the multithreaded execution *over the TLS code executed in a single core* (S_{comb}) is given by Equation 3.3. This performance variation reflects the combined effects of both ILP and TLP and the equality shown in Equation 3.4 holds.

$$S_{comb} = \frac{T_{1p}}{T_{mt}} \tag{3.3}$$

$$S_{comb} = S_{ilp} \times S_{tlp} \tag{3.4}$$

The performance variation of the multithreaded execution *over the TLS code executed in a single core* and that can be attributed to ILP effects (S_{ilp}) is given by Equation 3.5.

$$S_{ilp} = \frac{T_{1p}}{\sum T_i} \tag{3.5}$$

Thus, S_{ilp} can be computed with the measurements of T_{1p} and all T_is. The performance variation of the multithreaded execution *over the TLS code executed in a single core* and that can be attributed to TLP effects (S_{tlp}) can be computed by substituting the results of Equations 3.3 and 3.5 into Equation 3.4. The reason we compute S_{tlp} indirectly is that directly measuring overlap, especially in the presence of task squashes and re-executions, is very tricky and can lead to misleading results. Finally, we observe that the equality shown in Equation 3.6 holds, which shows that the final observed performance variation can be quantitatively attributed to the variations in the binary, to the ILP contributions, and to the TLP contributions.

$$S_{all} = S_{1p} \times S_{ilp} \times S_{tlp} \tag{3.6}$$

Comparing our model with that proposed in [63] it can be shown that the key difference is that the ILP estimate in that model is ultimately derived from the “dynamic

code bloat" factor f_{bloat}, while it is derived in our model from the actual instrumentation slowdown S_{1p}. The problem with using f_{bloat} as a proxy for the actual slowdown is that it implicitly assumes that the CPI of the unmodified sequential execution and that of the TLS-instrumented sequential execution are the same. In reality, however, the TLS instrumentation does affect the CPI as it involves the addition of mainly function calls and memory operations to set up thread spawns, which in turn have a different CPI from the rest of the thread. Obviously, this impact on CPI will be more pronounced for smaller threads than for larger ones. In our experiments (Section 3.4.1) we measured the difference in CPI and the ultimate impact on the ILP contribution estimation and found out that it can be significant in several cases.

3.4.1 Performance Model Comparison

The difference between our speedup breakdown model and that of [63] is that we propose to measure the actual execution time degradation of the TLS execution when running on a single core compared to the original sequential execution (S_{1p}), instead of estimating this factor with the instruction code bloat (f_{bloat}). In reality $1/S_{1p} \neq f_{bloat}$, which will lead to some inacurracy in the model of [63]. In fact, since the added TLS instrumentation consists of several memory operations and some function calls, we expect that $1/S_{1p} < f_{bloat}$ and the model of [63] will *under-estimate* the contribution of ILP.

We measured the difference between the ILP and the TLP estimates of both models for the baseline TLS system and show the results in Table 3.5. Although for some applications the errors are fairly small, this is not the case for some others like *gap* and *mcf* where there is a difference of 15.3% (for ILP estimation) and 13% (for TLP estimation). More importantly, in some cases the two models do not agree, so that the model proposed in [63] indicates that there is no ILP contribution (or there is a slowdown due to ILP degradation) whereas our model contradicts this. The benchmarks where this is the case are shown in Table 3.5 with bold. We thus believe that the extra simulations required by our model are well justified, since they provide a much clearer picture of what happens.

Table 3.5: Difference of ILP and TLP benefit estimation between our performance model and the one proposed in [63]. With bold we denote cases where our model presents speedup whereas the previously proposed does not.

Benchmark	**bzip2**	**crafty**	**gap**	**gzip**	**mcf**	**parser**	**twolf**	**vortex**	**vpr**	**avg**
% ILP Diff.	**1.81**	**0.45**	**15.3**	**2.67**	1.11	4.81	**2.26**	**2.97**	**3.49**	3.87
% TLP Diff.	5.09	7.19	0.02	2.89	13.0	9.32	5.98	6.64	2.01	5.79

Chapter 4

Combining TLS, HT, and RA Execution

4.1 Basic Idea

Each of the multithreaded execution models that we consider – TLS, HT, and RA – is best at exploiting different opportunities for accelerating the performance of a single-threaded application. We expect a combined scheme both to perform as well as the best model across a variety of applications and to even outperform the best model. The latter can happen if the combined scheme can adapt to the different acceleration opportunities of different program phases or if the acceleration opportunities are additive (e.g., if ILP can be exploited in addition to TLP for some program phase).

The basic idea of our proposal for combining TLS, HT, and RA execution is to start with a TLS execution and to convert some TLS threads into helper threads by switching them to runahead execution mode. Threads that have been converted to helper threads execute the same instructions as they would in TLS mode, but runahead execution is achieved by allowing them to predict the results of L2 misses instead of stalling, as done in the RA execution model (as opposed to being achieved by some compiler/programmer slicing mechanism). These converted helper threads can no longer contribute to the actual parallel computation (i.e., they can never commit) but can only help the remaining TLS threads execute more efficiently by prefetching for them in the shared L2 cache. In a multicore environment with TLS this can be achieved when the converted helper threads bring data into L2 that will be later used

by the remaining TLS threads. Note that in TLS the L1 cache is versioned so that no sharing and, thus, no prefetching, can occur across helper threads and TLS threads. Although traditional HTs are able to run ahead of the main thread and thus perform timely prefetches, in our case the same effect is achieved by making the threads faster (by allowing them to go past long latency misses). In both cases these threads help the other threads and as such we borrow the term HT and use it for our converted threads as well.

Combining TLS, HT, and RA execution is a reasonable approach for three main reasons. Firstly, because we start by employing only TLS, we first try to extract all the available TLP. This makes sense in a system with several cores since, if TLP can be exploited, it is more likely that this will yield better performance than trying to speed up a single main thread. When we fail to speculatively extract TLP, we may utilize the extra hardware resources to improve the ILP of the main thread, whereas a base TLS system would be idle. Secondly, accommodating HT and RA execution within the TLS execution model requires only slight modifications to the base TLS system (Table 2.1). Finally, starting from TLS threads and speeding them up using the RA model is a simple and automatic way of generating helper threads (no programmer intervention is required).

While the basic idea is simple, developing a fully working system requires dealing with a few implementation issues. The key issue relates to the policy of *when, where and how to create HT*. These decisions are critical because in our HT/RA model threads do not contribute to TLP and consume TLP resources (e.g., cores, caches), so that a conversion policy must balance the potential increase in ILP with the potential loss of TLP. Another aspect of this is whether helper threads are simply converted from existing TLS threads in place, or whether new TLS threads are specifically created elsewhere for the purpose of becoming helper threads. This decision also affects how to manage helper threads in the context of an extended TLS environment. In particular, the TLS environment imposes a total ordering on the threads in the system, which is reasonable for TLS threads, but becomes slightly more involved when some threads are TLS and some are HT. Also, a question is what to do with a helper thread when it detects a data dependence violation and when one of its predecessors is squashed.

These issues are discussed in Section 4.2. Finally, another important issue relates to the policy of converting threads back from HT to TLS threads. Since our simplified HT/RA model does not allow for their execution to be integrated into the TLS execution, this latter policy boils down to *how to destroy HT* and free up resources for further TLS threads. This issue is discussed in Section 4.3.

4.2 When, Where and How to Create HT

We identify two suitable occasions for creating a helper thread: at thread spawn and when a TLS thread suffers an L2 cache miss. By creating a helper thread on every thread spawn, we make the assumption that the original thread will benefit from prefetching, which may not be always true. On the other hand, creating a helper thread on an L2 miss will only help if the original thread will suffer more L2 misses later. Luckily we find that TLS threads exhibit locality in their misses, that is, they either suffer many misses in the L2 cache or they do not suffer any (due to changes in the working set some iterations of loops have a lot of misses. Subsequent iterations of the same loop find the data in the cache due to temporal/spatial locality). We experimented with both approaches and found out that indeed this was the case and that the approach of spawning helper threads on a L2 miss performs better (Section 5.3.1).

As for the location where to execute the helper thread there are two possibilities: in the same core where the original TLS thread was executing (thus, effectively putting the TLS thread into runahead mode) or in a different idle core (thus, effectively *cloning* the original TLS thread and converting the clone into a helper thread, see Figure 4.1(a)). Obviously, the first option will sacrifice the exploitation of TLP, which may not be easily recovered by the benefits of the helper thread. On the other hand, the second option leads to an increased number of threads in the system, which increases the pressure on resources, possibly leading to performance degradation. If we decide to convert an existing thread, we simply have to *checkpoint* and mark the thread as a helper thread. This thread will proceed until the end of its execution disregarding long latency events and restart signals. If we instead create a new thread, we will have to do so using the existing TLS spawning model and marking the thread as a helper thread.

We experimented with both approaches and found out that the latter performs better (Section 5.3.2).

Throughout this work we optimistically assume that TLS threads will perform useful work. Since helper threads require resources that could otherwise be used by TLS threads, indiscriminately creating helper threads at every L2 cache miss may prove detrimental to the final system's performance. Thus, it is important to make the helper threads as transparent as possible. A simple way of doing this is by allowing only a small number of helper threads to exist at any given time and to ensure that TLS threads will always be given priority over these helper threads. Although spawning on an L2 miss will create helper threads only for the threads that will likely benefit from prefetching, this may still result in the creation of too many threads. For this reason we only create a helper thread for the L2 misses if there is a free processor. Additionally, we do not allow any of the helper threads to perform any thread spawn themselves. In all cases, if we spawn a normal TLS thread and we do not have any free processor available, we pre-empt one of the running helper threads by killing it. We experimented with these different approaches and found out that keeping the number of helper threads small with the policies above gives better results (Section 5.3.3).

We also found that by allowing only the most speculative thread to spawn a helper thread on an idle core we can achieve most of the benefits one can achieve by allowing multiple helper threads to co-exist (Section 5.3.4). This is to be attributed to a combination of deeper prefetching and a *chain-prefetching* effect. Deeper prefetching is the result of using as helper threads TLS threads that are far ahead in execution when compared to the least speculative ones. With chain prefetching we refer to a phenomenon under which a helper thread prefetches only for the most speculative thread, which in turn goes faster and prefetches for its parent thread. An example of this can be seen in Figure 4.2, where under normal TLS (Figure 4.2(a)) both *Thread 1* and *Thread 2* suffer L2 misses at about the same time. When we clone *Thread 2*, we manage to get rid only of the second miss from *Thread 2* (Figure 4.2(b)). However, this makes *Thread 2* reach the third miss faster and, thus, prefetch it for *Thread 1*.

In addition to throttling the use of resources, another reason for only allowing the most speculative thread to spawn a helper thread is that it greatly simplifies the com-

bined TLS/HT protocol. By doing so, we separate the TLS and the helper threads in the total thread ordering scheme. This in turn means that we do not have to deal with complex interactions, such as the case that a helper thread triggers a data dependence violation with a more speculative TLS thread or that a more speculative TLS thread attempts to consume a version produced by the helper thread.

4.3 When to Terminate a HT

A helper thread should be terminated in any of the following five cases. The first case is when the helper thread reaches the commit instruction inserted by the compiler that denotes the end of the thread. The second case is when the parent thread that created the helper thread reaches the commit instruction (Figure 4.1(a)). The third case is when the parent thread finds a TLS spawn instruction (Figure 4.1(b)). Fourth, if the thread that created the helper thread receives a restart or kill signal, the helper thread has to be killed as well to facilitate keeping the ordering of threads in the system (Figure 4.1(c)). Finally, helper threads use predicted values for the L2 cache misses and as such they might follow incorrect execution paths. So the fifth case occurs when one of these paths causes an exception.

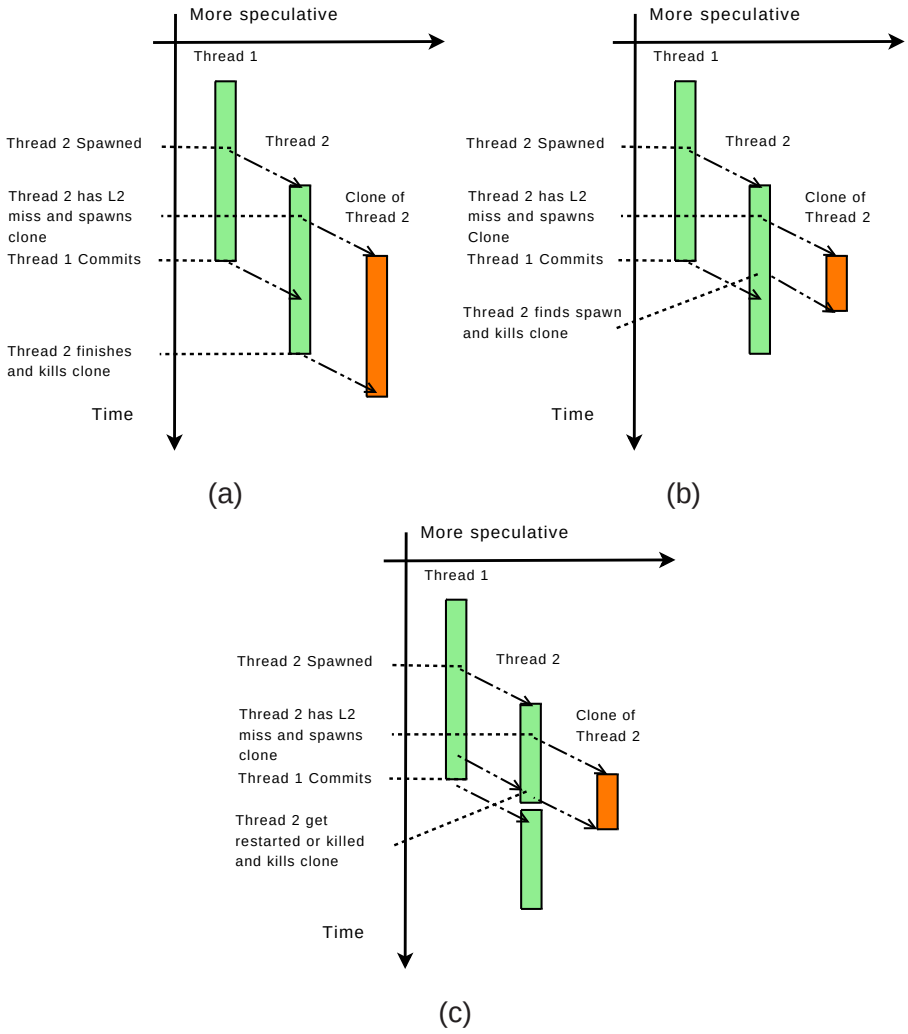

Figure 4.1: Helper threading with clones: (a) A thread is cloned on an L2 miss. (b) The clone is killed on a thread spawn. (c) The clone is killed on a restart/kill of the thread that spawned it.

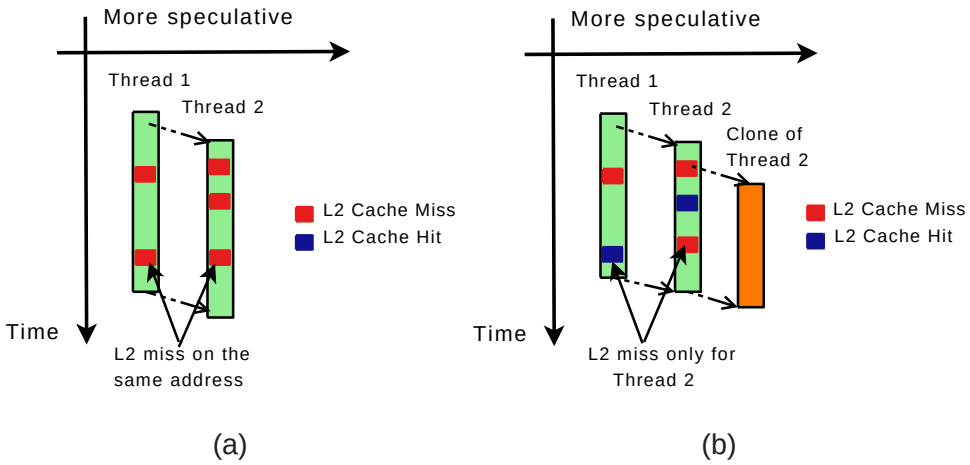

Figure 4.2: Chain Prefetching effect: (a) Under normal TLS execution both threads find L2 cache misses concurrently. (b) The clone prefetches for Thread 2, which in turn prefetches for Thread 1.

Chapter 5

Analysis of the Combined TLS/HT/RA Scheme

5.1 Additional Hardware Configuration to Support HT and RA

Our combined TLS/HT/RA scheme requires on top of the baseline TLS support a value predictor so as to predict the value to be returned by missing loads. Throughout most of the evaluation we use a simple last-value predictor [48], but we show later that a better value predictor could improve the overall performance significantly. The value predictor is address based and it is updated on every L2 cache miss. Our scheme also requires to transfer register state on a spawning of a clone thread. This is implemented using microcode and it adds an additional cost to the creation of a clone thread of 100 cycles, by pessimistically assuming 1 cycle per register transfered. Note that normal TLS thread spawn only requires 20 cycles, since register state transfer is done through memory (the compiler inserts spills to memory). Techniques to speed-up the spawn process would yield even better results, but as I will show in the later sections our techniques do not rely on them.

The main microarchitectural features are listed in Table 5.1. The top part lists the baseline parameters as shown in Table 3.1, and the bottom part lists the additional parameters.

Table 5.1: Architectural parameters with additional parameters required for HT/RA support.

Parameter	**TLS (4 cores)**
Fetch/Issue/Retire Width	4, 4, 4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80, 104
Ld/St Queue	54, 46
Branch Predictor	48Kbit Hybrid Bimodal-Gshare
BTB/RAS	2K entries, 2-way, 32 entries
Minimum Misprediction	12 cycles
Task Containers per Core	8
Cycles to Spawn	20
Cycles from Violation to Kill/Restart	20

Extra Hardware per Core	
Value Predictor	4K entries, Last-Value

We start by showing the bottom-line results of our scheme when compared with both TLS and a flavor of runahead execution that uses value prediction but always reverts to the checkpoint made before going into Runahead mode (i.e., it discards all computation done in the Runahead thread and, thus, does not exploit any TLP). We next try to quantitatively explain how our scheme works and provide a detailed analysis of the reasons that led us to the proposed design. We do not compare against pure HT because it involves significant compiler support and there is no single representative scheme of HT.

5.2 Comparing TLS, Runahead, and the Combined Scheme

5.2.1 Performance Gains

Figure 5.1 shows how our proposed scheme performs when compared with both TLS and runahead execution with value prediction. Each of the bars shows the total speedup and the proportion of the speedup that can be attributed to ILP and TLP based on the methodology discussed in Section 3.4. Speedups are relative to sequential execution with the *original* sequential binary. With the light grey shade below the 1.0 point we denote the base case each of the schemes starts from when running on a single core (TLS and combined scheme have worse quality of code; this is the S_{1p} factor of Section 3.4) and with the next two shades the proportion of speedup due to ILP and TLP accordingly [1]. The leftmost bars correspond to the base TLS, the middle bars to runahead execution with value prediction, and the rightmost bars to our combined scheme.

Considering the base execution models alone, we first note that while TLS performs better than runahead execution for most applications, for some applications the performance of both schemes is comparable (*gzip* and *parser*). It is interesting that for the memory bound *mcf* runahead execution is able to outperform TLS, even though it uses only one core. On the other hand for two applications, namely *gap* and *vortex*, runahead suffers a slowdown (for this reason the grey shades of the corresponding bars

[1] Note that the breakdown shows proportions of speedup due to each category and the height of each portion cannot be directly read as the speedup coming from ILP and TLP.

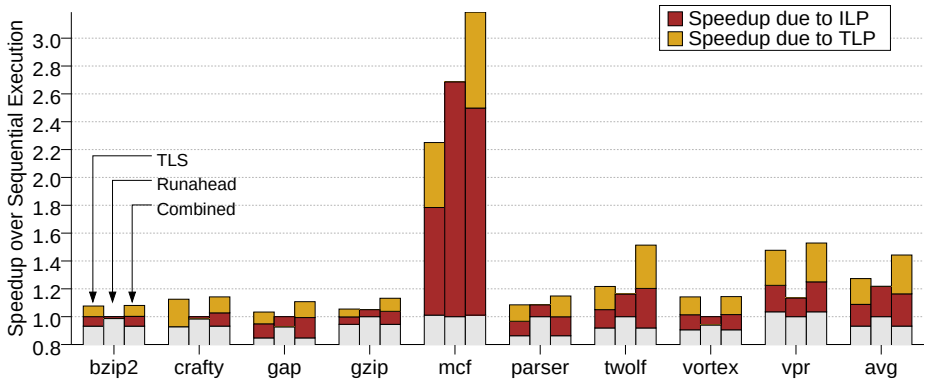

Figure 5.1: Speedup breakdown based on our performance model. Leftmost bar is for TLS, middle bar is for runahead with value prediction, and rightmost bar is for our combined approach.

start from below 1.0), due to cache pollution. We also note that, despite the main target of TLS being the exploitation of TLP, for most applications the benefits of ILP in the base TLS scheme are comparable to the TLP benefits. For the same set of applications state-of-the-art parallelizing compilers like Intel's ICC, are not able to achieve any speedups (in fact they slow down some of the applications)

Comparing our combined scheme with TLS and runahead, we see that the combined scheme performs at least as well as the best of the other schemes for all applications and often outperforms the best scheme. The performance advantage of the combined scheme indicates that often the speedups of runahead execution and TLS are additive. In fact, even in applications where runahead execution fails to deliver any speedups (*crafty, gap* and *vortex*), our combined scheme achieves speedups equal to or better than TLS. When compared with TLS, we see that our combined scheme obtains greater performance gains from ILP while maintaining most of the TLP benefits. Interestingly, in some cases our combined scheme is better than the base TLS scheme even in terms of TLP. One possible reason for this is that faster speculative threads will uncover violating reads faster and thus perform restarts earlier. This is an effect similar to having lazy or eager invalidations, where it is known that eager invalidations procure better results due to increased TLP. When compared with runahead execution, we see

that our combined scheme obtains gains from TLP while maintaining most of the ILP benefits. Again, in some cases our combined scheme leads to higher ILP benefits than the base runahead execution. The likely reason for this is the deeper prefetching effect that can be achieved by performing runahead execution from a speculative thread, coupled with the chain-prefetching effect described in Section 4.2. Overall, we see that our combined execution model achieves speedups of up to 41.2%, with an average of 10.2%, over the existing state-of-the-art TLS system and speedups of up to 35.2%, with an average of 18.3%, over runahead execution with value prediction.

5.2.2 Cache Miss Behavior

Figure 5.2 depicts the number of L2 misses on the *committed path* for all the schemes normalized to the sequential case. This figure allows us to quantify the amount of prefetching happening in each scheme. Note that all three schemes have smaller miss rates than sequential execution on average since all of them perform some sort of prefetching. Runahead execution leads to only a relatively small reduction in L2 misses and, in fact, for some applications like *parser* and *twolf* it actually slightly increases the number of L2 misses. For TLS prefetching is slightly less than runahead, and for some applications (like *mcf*, *parser* and *twolf*) TLS suffers from more misses than the sequential execution. This happens due to code bloating and the overall lower quality of the code produced (some compiler optimizations are restricted by the TLS pass). The combined scheme on the other hand, is able to prefetch significantly more useful cache lines reducing the miss rate by 41% on average when compared with the miss rate of sequential execution.

Figure 5.3 shows the fraction of isolated and clustered misses seen *on the committed path* for the various execution models. We define a miss as an isolated miss, if when it reaches the Miss Handling Registers (MSHR) the cache line is not already being serviced by a previous miss to the cache. Clustering is identified as the presence of other in-flight memory requests when the commit path suffers another L2 cache miss. This figure is then complementary to Figure 5.2 in explaining the prefetching effects of each model as it can capture *partial* prefetches (i.e., prefetches that do not completely eliminate a cache miss, but lead to a reduction in the waiting time). Note that runa-

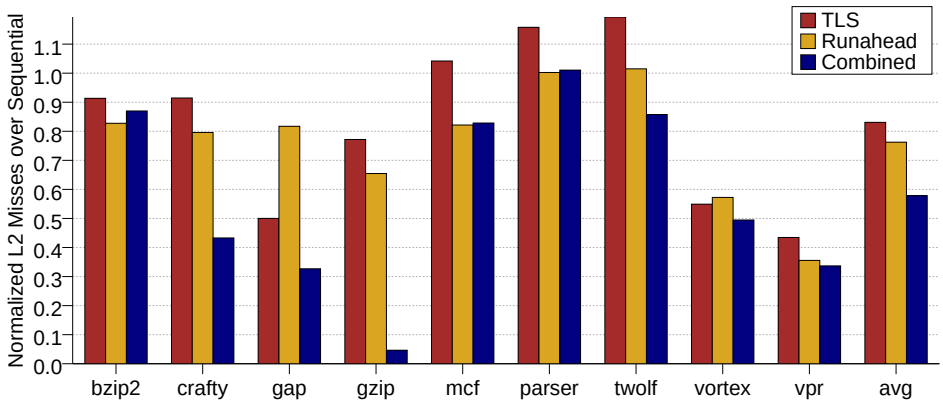

Figure 5.2: Normalized L2 misses over sequential execution for the committed path of TLS, runahead execution, and our combined scheme.

head execution does very well in clustering the misses and in fact is able to do much better than both TLS and our combined scheme. However, as noted above, this significant increase in number of outstanding memory requests with runahead execution does not always translate into fewer L2 misses seen by the commit path (Figure 5.2), but in some cases it does lead to partial prefetches and improved ILP (Figure 5.1). Our combined scheme manages to cluster the misses much better than TLS does, leading to further benefits from partial prefetches.

5.3 Understanding the Trade-Offs in the Combined Scheme

5.3.1 When to Create a HT

As we discussed earlier in Section 4.2, there is a choice to be made whether to create a new helper thread on an L2 cache miss or at thread spawn time. Figure 5.4 shows the speedup of each of the two policies. As the figure clearly shows, cloning on L2 misses is always better. The reason is that it is more targeted and, thus, it does not increase the number of running threads unless there are prefetching needs (i.e., at least one actual L2 miss). This is evident from the ILP/TLP breakdown where we see that the main difference between the two policies is mostly in the ILP benefits.

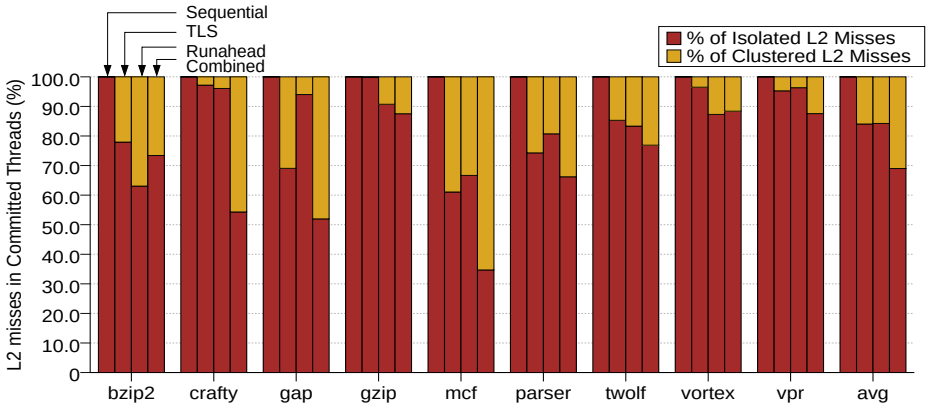

Figure 5.3: Breakdown of the L2 misses in isolated and clustered for sequential, TLS, runahead, and our combined scheme.

5.3.2 Converting Existing Threads into HT vs. Spawning New HT

An interesting trade-off is whether one should try to convert some of the normal TLS threads to helper threads (checkpointing), or whether one should create separate helper threads (cloning). As we discussed in Section 4.2, converting some TLS threads to helper threads will increase the ILP but it will do so at the expense of the TLP we can extract. Figure 5.5 compares the two policies. As the figure shows, spawning a new helper thread leads to better performance in all but one case (*crafty*). It is interesting to note that in most cases the performance advantage of the cloning approach comes not only from increased TLP, as one would expect, but also from increased ILP. It is also worth noting that for the converting approach, although in all of the cases the ILP gains are significant, for some benchmarks this policy performs even worse than the base TLS.

5.3.3 Effect of the Load of the System

Helper threads require resources that could otherwise be used by normal TLS threads. Figure 5.6(b) shows the average distribution of number of threads that exist at a given time in a four core system with TLS (for the whole applications, including sequential parts). We can see that almost 90% of the time there are only up to two threads running.

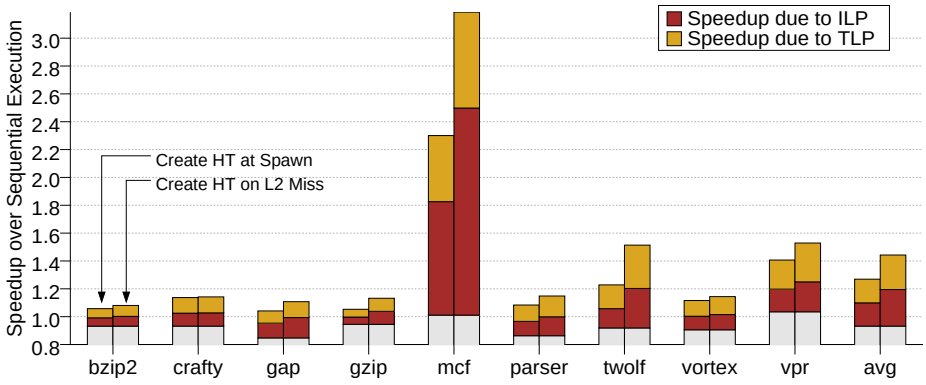

Figure 5.4: Impact of choosing between creating helper threads on an L2 miss or at thread spawn.

This means that as long as we create a small number of helper threads we should not harm the extracted TLP. This result also suggests that having more than four cores, will probably not make any difference for our baseline TLS system.

In Figure 5.6(a) we first show a helper thread spawning policy under which we create a new helper thread on every L2 cache miss. Under this scheme we allow multiple helper threads to co-exist. This scheme does not check if there is any available core to run the new helper thread on (if there is no free core, threads are placed in queues waiting to execute as we do with normal TLS spawns), and the clones are not killed when we spawn a new TLS thread. We compare this with our load-aware scheme, under which we only allow one helper thread to exist at a time and we kill helper threads in order to pre-empt them. The benefits of employing a load-aware scheme are more pronounced in applications with a large number of threads like *mcf* and *twolf*. The benefits come mainly from better ILP since, we are making the contention on the common L2 cache smaller and we are poluting it less. Figure 5.6(c) depicts the utilization for the two different policies. As expected there is a slight shift towards having more threads running when the system is not aware of the system load.

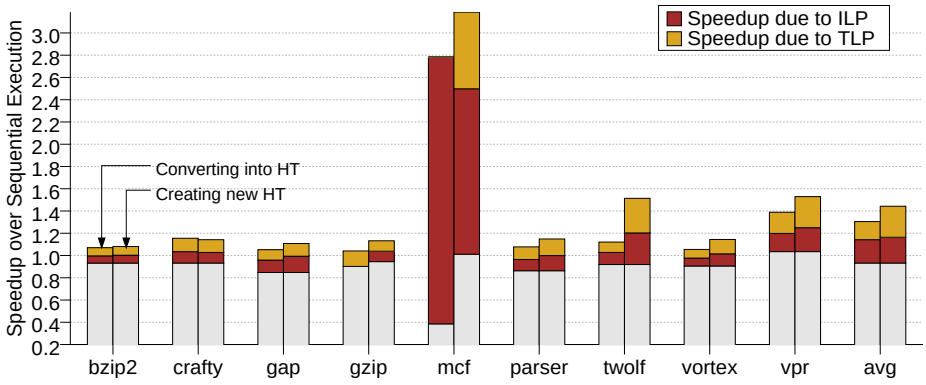

Figure 5.5: Converting existing TLS threads to helper threads and spawning distinct helper threads.

5.3.4 Single vs. Multiple HTs

In our approach, we chose to create a single helper thread by cloning the most speculative thread because of its fairly straightforward implementation overhead and reasonable expected performance benefits. We compare our scheme with two other schemes: one where we only allow the safe thread to create a clone, and one where we allow multiple clones to run in the system. Note that both schemes respect our load-aware policies (Section 5.3.3) so that they do not interfere negatively with the normal TLS threads. This means that for our four core system we can have at most two normal and two clone threads running concurrently (as opposed to our scheme where we will only have one clone thread).

As Figure 5.7 shows, cloning only for the safe thread gives only a fraction of the achievable benefits. This is mainly due to worse ILP, which makes sense if we take into account that we are only prefetching for one thread. In fact, Figure 5.8 shows that creating a helper thread for the most speculative task performs substantially more useful prefetching than the scheme where we only create a helper thread for the safe thread. On the other hand, creating a clone for all the threads is slightly better than our scheme for all applications except *mcf*. Figure 5.9 shows the clustering of memory requests for all three schemes. In most cases the difference in clustering is not very significant. However, the clustering helps explain the case of *mcf* with our scheme and

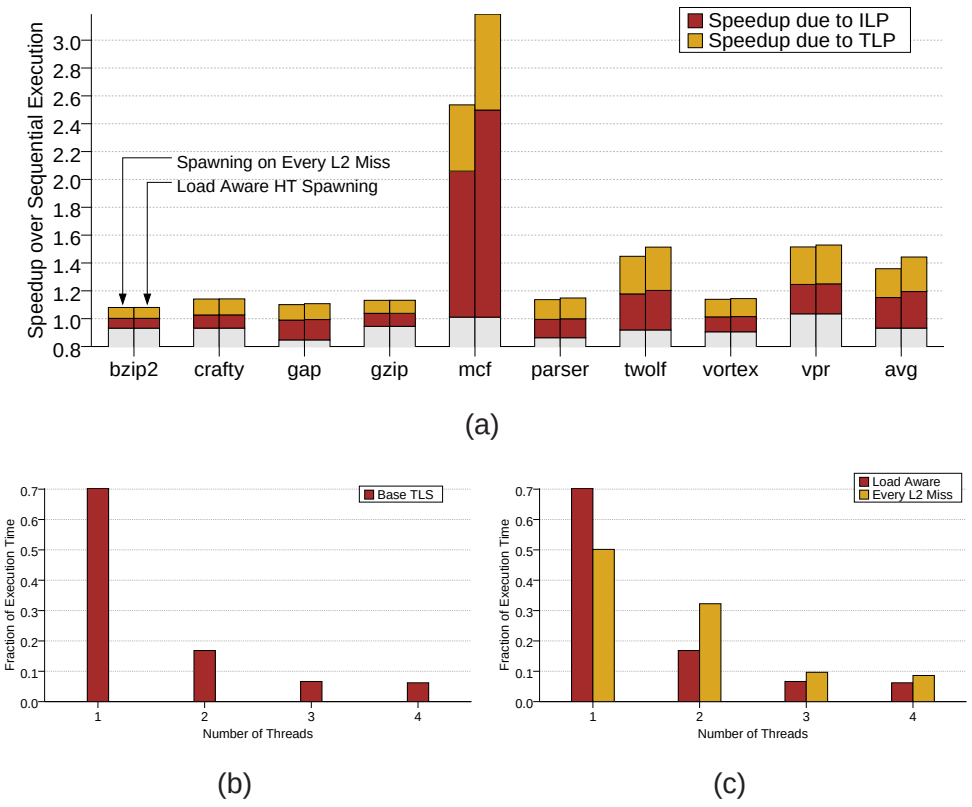

Figure 5.6: (a) Evaluating the effect of load aware HT spawning. (b) Average distribution of number of threads on a four core system across the Spec 2000 Integer Benchmarks. (c) Average distribution of number of thread for load aware and load unaware schemes across the Spec 2000 Integer Benchmarks.

with cloning for all threads: even though the miss rates are comparable (Figure 5.8), the effect of partial prefetching is much more pronounced with the combined scheme.

5.3.5 Effect of a Better Value Predictor

Throughout our study we have employed a simple last value predictor for the runahead helper threads. In this section we perform a sensitivity analysis of our scheme on this building block. As we see in Figure 5.10, a better value predictor (i.e., a perfect one in this case) would lead to significantly increased benefits for the combined scheme.

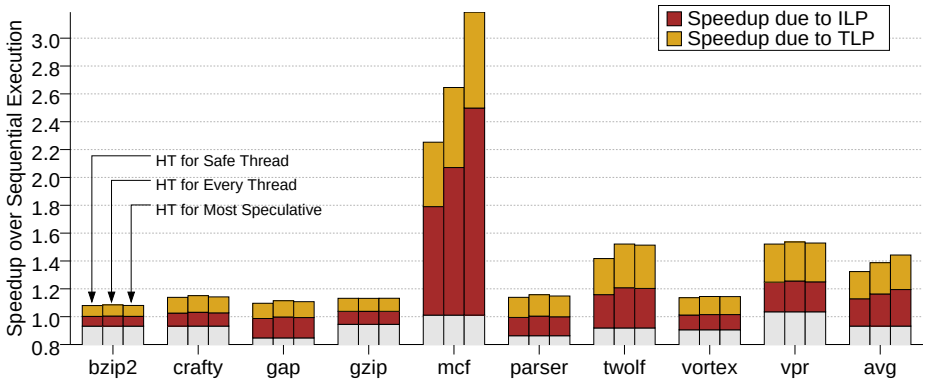

Figure 5.7: Comparing the effect on performance when creating multiple HT or a single HT.

The improvement will come in terms of better ILP, which is justified by the improved prefetching capability resulting from always following the correct path. We see that there is still ample room for improvement for at least four out of the nine benchmarks used in this thesis. Figure 5.11 shows that the improvement in ILP comes from a significant reduction in the achieved miss rates of the level two cache. The reason is that the helper threads that we create has a higher probability to follow the correct path, and as such always perform useful prefetches. Figure 5.12 shows that the number of isolated misses are also reduced significantly. This result suggests that there are many branches that are data dependent on values that our last value predictor tends to mispredict.

5.4 Sensitivity to Microarchitectural Parameters

5.4.1 Using a Prefetcher

A valid point of concern with all the schemes that perform prefetching/warming-up is whether a traditional hardware prefetcher could perform better. Figure 5.13 shows what the effect of having an aggressive stride prefetcher in our base cores is. All comparisons are with sequential execution on a core that uses the same prefetcher.

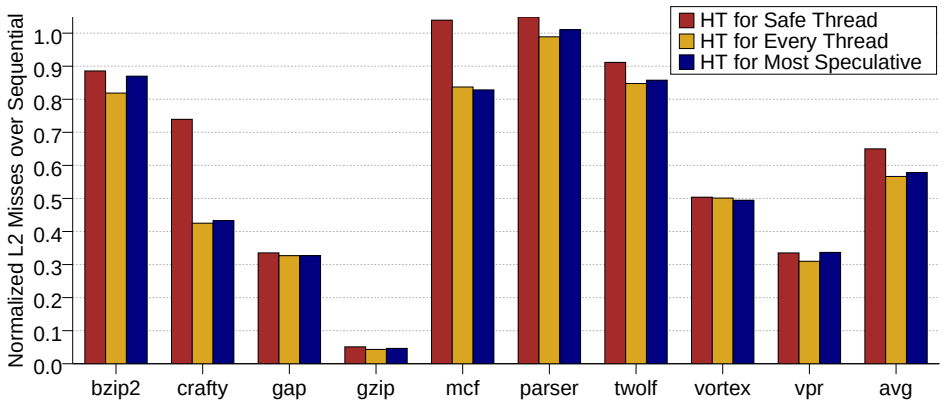

Figure 5.8: Normalized L2 misses over sequential execution when creating multiple HT or a single HT.

Note that although our combined scheme is better than TLS for all of the applications, when we add a prefetcher to TLS this changes for two applications, namely *gap* and *parser*. Note, however, that when we use a prefetcher with our combined scheme, not only are we able to perform better for all applications except *mcf*, but we are able to perform better in applications where our combined scheme failed to perform significant prefetching (for *gap* and *parser*). As it has been shown by previous research on runahead execution [58], we believe that this is due to our Runahead threads initiating prefetches earlier than they would have been initiated otherwise, and as such more timely prefetches.

Figure 5.14 shows that when we compare the combined scheme with a combined scheme that has hardware prefetching support, there is further L2 miss reduction from the hardware prefetcher. Note that for the benchmarks where the prefetcher helps the performance of the base combined scheme (*gap* and *parser*) there is a significant improvement in terms of miss rate. It is interesting to see that for *mcf*, which is the only benchmark for which the base combined scheme was better, the miss rate improves. Unfortunately, the prefetcher along with our aggresive scheme causes contention for the shared cache, which results in a slight degradation in performance, although the miss rates are better.

Figure 5.15 shows that on average the percentage of isolated misses is larger for

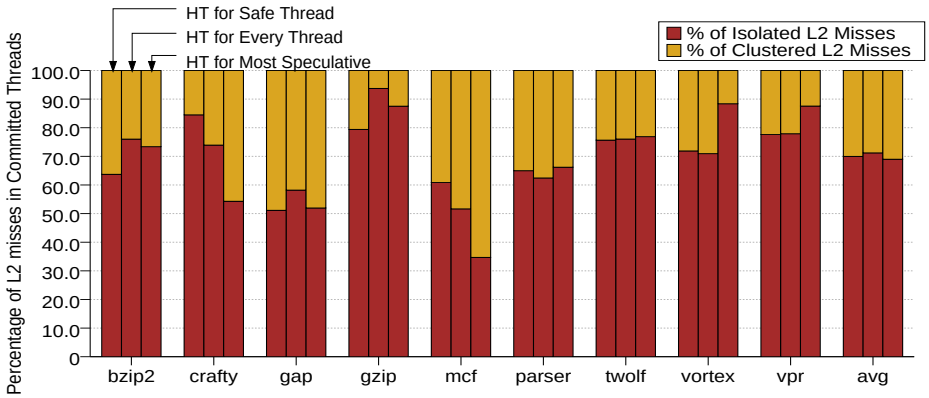

Figure 5.9: Breakdown of the L2 misses in isolated and clustered when creating multiple HT or a single HT.

the case where we use the prefetcher. This somewhat strange result, when seen in conjuction with the improved miss rate shown in the previous figure, suggests that the prefetcher helps not by performing prefetches our scheme would not perform otherwise, but by performing them in a more timely way. More specifically, when a prefetcher is used, clusters of misses are slightly fewer because the lines have been prefetched and can be found in the cache (thus some of the misses that create a cluster without prefetching are converted to hits with prefetching).

5.4.2 Scaling the Memory Latency

It is well known that the gap between the speed of the core and that of the memory keeps growing. Figure 5.16 shows how our combined scheme will behave if this gap doubles. As expected the combined scheme performs significantly better than the sequential execution with the longer memory latency, since it is able to extract even more ILP. On average the achieved speedup increases by 33% as a result of the increase of the memory latency, a fact that denotes that if the memory gap continues to widen, this technique will become increasingly important.

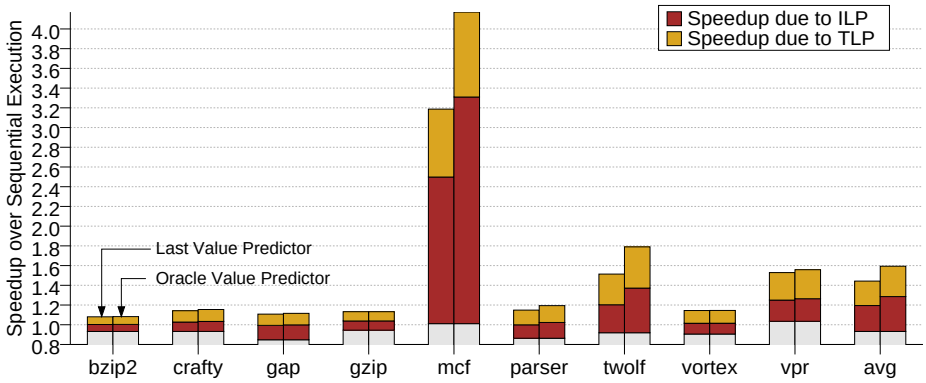

Figure 5.10: Performance impact of value prediction accuracy.

5.4.3 Scaling the Available Bandwidth

As we move to a higher number of available cores, we expect the available bandwidth per core to lessen. Although in most cases bandwidth is dynamically allocated, we performed a simple experiment to see what effect a reduction in bandwidth will have in our scheme. More specifically, we halved the available bandwidth for both the sequential execution and the proposed combined scheme and compared it with the combined scheme and the sequential execution with the normal bandwidth. Figure 5.17 shows that even when we reduce the available bandwidth quite significantly we do not see significant reductions in the achieved speedups, with the exception of *mcf*. Note, however, that even under this fairly pessimistic scenario the achieved speedup over the base execution models is still significant (TLS achieved 27% speedup over sequential).

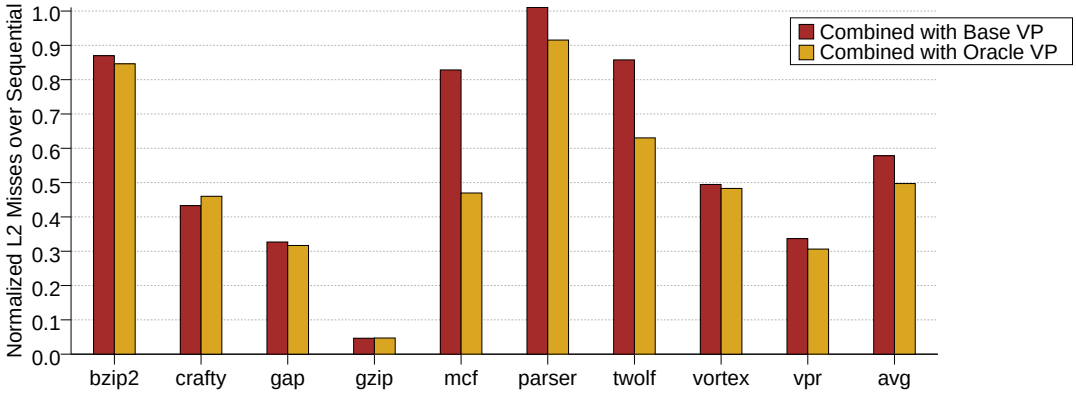

Figure 5.11: Normalized L2 misses over sequential execution for the combined scheme with the base value predictor and with an oracle one.

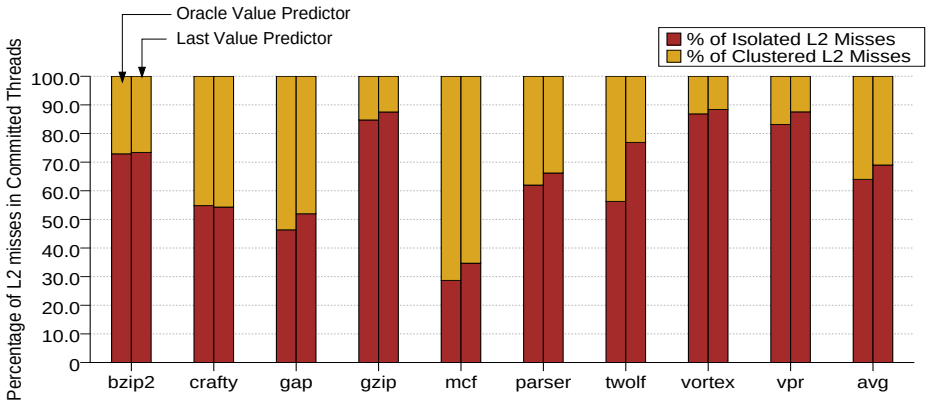

Figure 5.12: Breakdown of the L2 misses in isolated and clustered for the Combined scheme with the base value predictor and with an oracle one.

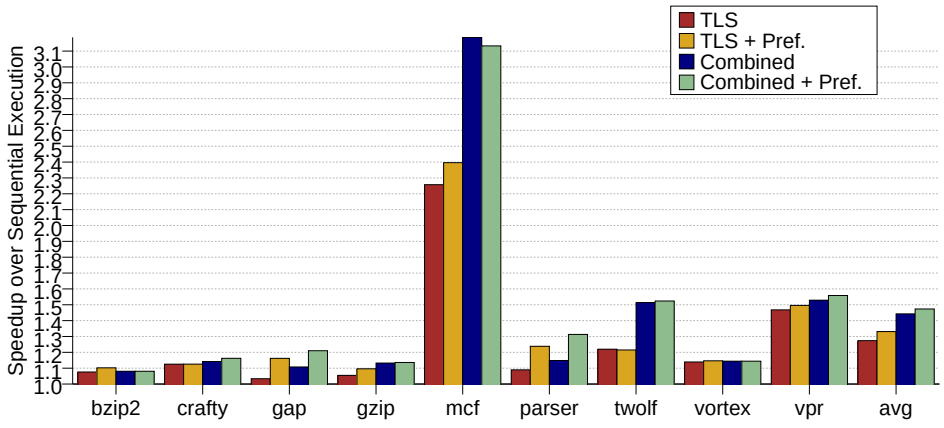

Figure 5.13: Performance of TLS and our combined scheme with and without a prefetcher (the baseline sequential execution uses the same prefetcher).

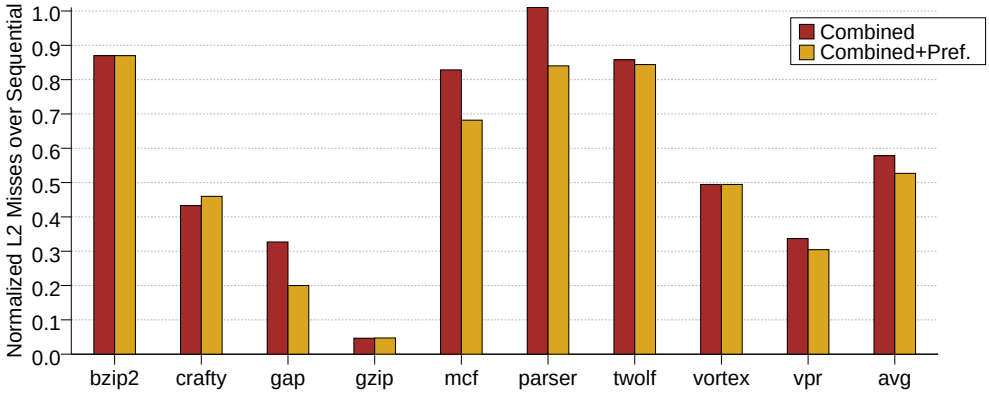

Figure 5.14: Normalized L2 misses over sequential execution for the combined scheme and the Combined scheme with a stride prefetcher.

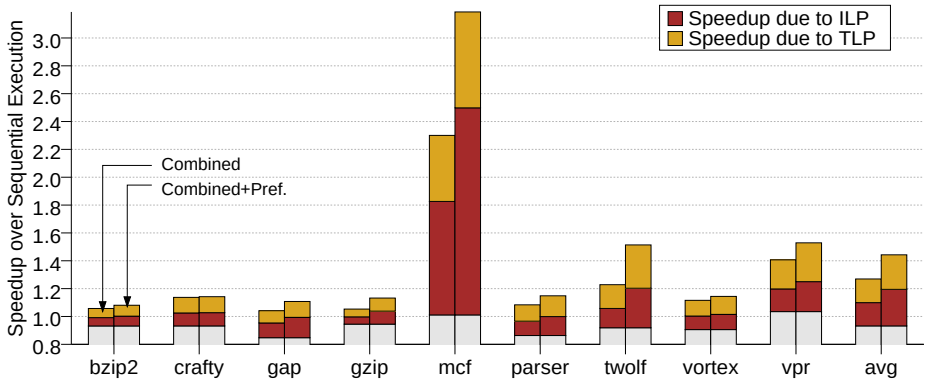

Figure 5.15: Breakdown of the L2 misses in isolated and clustered for the combined scheme and the Combined scheme with a stride prefetcher.

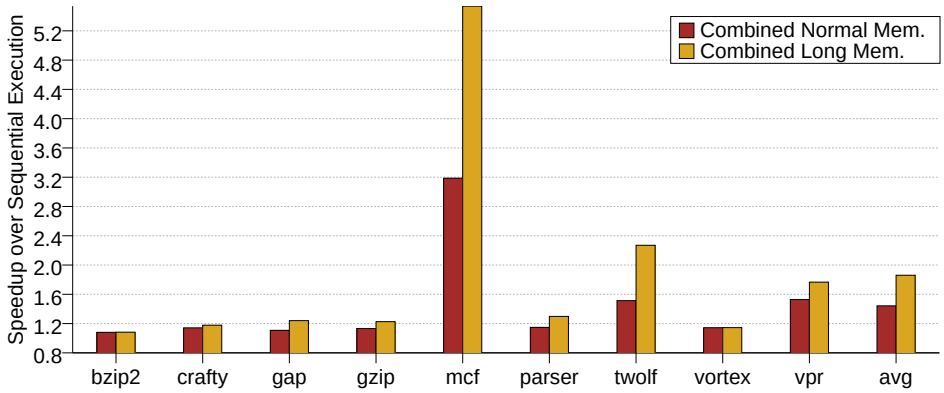

Figure 5.16: Combined scheme for the normal main memory latency of 500 cycles, and for a latency of 1000 cycles.

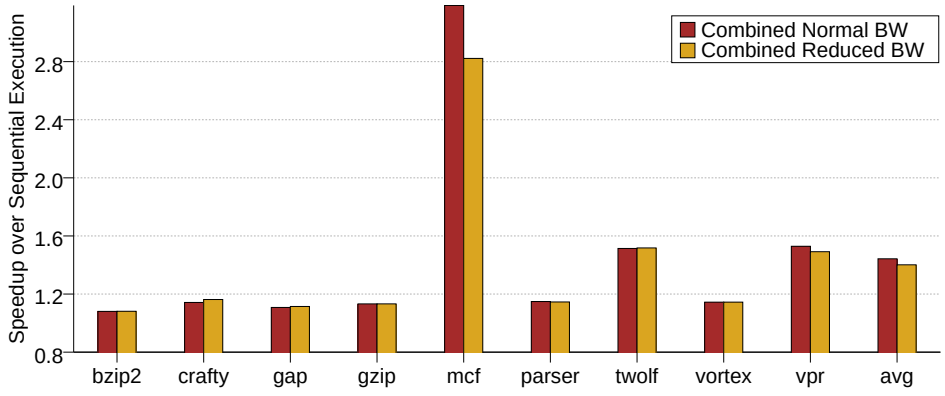

Figure 5.17: Combined scheme for the normal bandwidth (10 GBytes/sec), and for half the normal bandwidth (5 GBytes/sec).

Chapter 6

Analysis of Branch Prediction in TLS

6.1 Impact of Branch Prediction on TLS

We start our analysis by attempting to verify the importance of branch prediction accuracy on the overall TLS performance. For this purpose, we replace the branch predictor in each processor with an oracle predictor so that we can artificially set any desired misprediction rate. More specifically, our oracle predictor causes mispredictions based on a random distribution with a mean with the desired misprediction rate. Details regarding the exact micro-architectural parameters used for this analysis are provided in Section 3. Figure 6.1 shows the relative performance improvement of the sequential execution and TLS executions with 2, 4, and 8 processors, as the branch misprediction rate decreases from 10% to 0 (i.e., perfect branch prediction). Each line is normalized to the performance of the *same configuration* with branch prediction with a misprediction rate of 10% and corresponds to the geometric mean of the performance improvement of all benchmarks. From this figure we can see that branch prediction accuracy can have a significant impact on TLS performance. In fact, it seems that better branch prediction accuracy can be more important for TLS performance than for sequential performance. For instance, reducing the branch misprediction rate from 10% to 2% improves the performance of TLS on 8 processors by 38%, but improves the performance of sequential execution by 32%. Note also that the more cores we assign to TLS the more important branch prediction is for it. We observed that the main reason

for this increased sensitivity of TLS to branch prediction accuracy is the increased average observed misprediction penalties in the TLS case. As has been previously noted in [29] branch misprediction penalty is directly related to observed memory system latency when branches depend on load values. As memory system latencies are usually larger with TLS than with sequential execution due to version misses, the average branch misprediction penalties are higher.

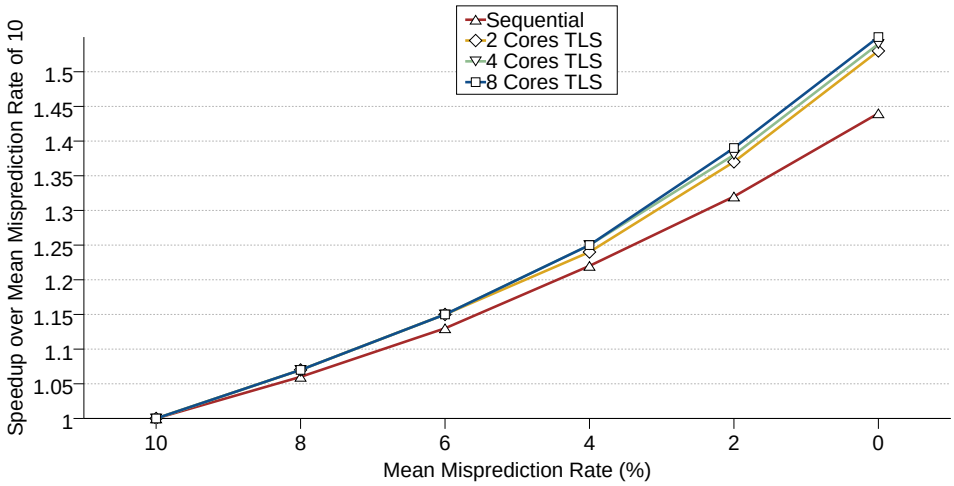

Figure 6.1: Normalized speedup with varying misprediction rate for sequential execution and for TLS systems with 2, 4, and 8 processors. Speedups are normalized to those at 10% branch misprediction rate on the same configuration.

6.2 Quantifying Traditional Branch Prediction Enhancing Techniques

Having shown how important branch prediction is for TLS systems, we will now quantify how suited conventional ways to increase the branch prediction accuracy are for such systems. More specifically, we will show what the impact is on misprediction rates of using a better branch prediction algorithm (which removes mispredictions due to better disambiguation of branch behavior) and what the impact is of making predic-

tors larger (which removes mispredictions caused by aliasing in the history tables).

We start by showing what the impact is of creating a better branch predictor. For this experiment we compare the misprediction rates of a single core with that of a four core system for different branch predictors of the same size. In Figure 6.2 we see for the two cases the misprediction rates as we progress from a bimodal predictor, to a gshare [55], to a hybrid (bimodal/gshare) [55], to a BcGskew [65] and finally to a state-of-the-art OGEHL [66] (all of them have a constant 32Kbit budget and we report numbers only for the committed path for TLS). Note that predictors with no history (bimodal) are far worse for TLS systems than they are for the sequential ones. The reason for this is that when data speculation fails (a fact that will cause a squash later on) the predictor follows a path different than what it would if data speculation did not fail. When the thread receives a restart signal, the predictor may be trained based on the wrong paths and as such mispredict. The picture is different for predictors with small histories like the gshare and the hybrid (bimodal/gshare), where they perform better than they do for single threaded systems. The reason for this is that due to squashes and re-executions, predictors are trained for sub-sequent executions. Unlike the bimodal predictors, these predictors are able to disambiguate branches based on the control-flow-graph followed, so that wrong path training due to data dependent branches does not affect prediction accuracies. Unfortunately for predictors able to exploit large histories like BcGskew and OGEHL, the history partitioning with TLS hurts the prediction accuracies beyond this benefit.

Unfortunately, history partitioning is inevitable for TLS systems and is a result of the sequential code being partitioned into threads and distributed among the available cores. Intuitively, history partitioning should reduce the branch predictors' accuracy, since the information on which the predictors rely to make their predictions is reduced with increase in the number of cores in the system.

This means that traditionally creating predictors with larger history registers is likely to be less efficient for TLS systems than it is for sequential ones. This has been previously also reported in [20]. For this reason that work proposed the use of history register initialization techniques. Our reported numbers already use that technique (without it, the picture is far worse for TLS).

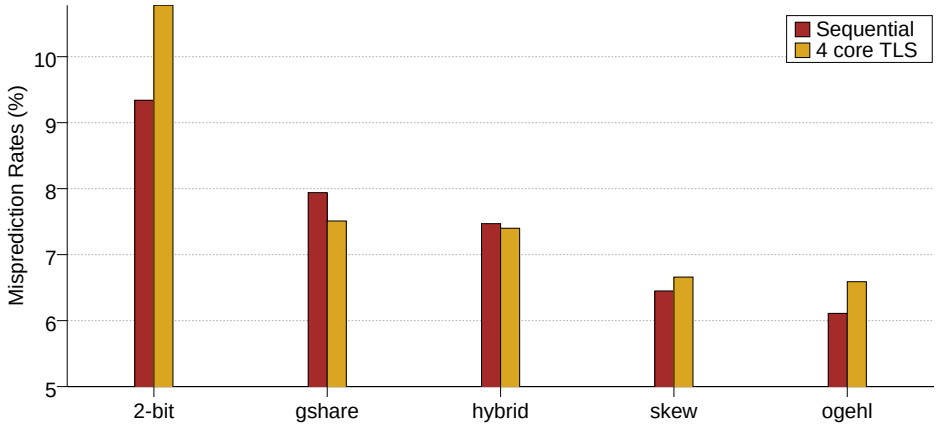

Figure 6.2: Improvement in misprediction rates for TLS execution on 4 processors and sequential execution for varying predictor types.

Figure 6.3 shows how the misprediction rate varies with predictor size for a TLS execution on 4 processors and for sequential execution for the OGEHL branch predictor. As we can see from this figure, TLS does seem to benefit less from larger predictors than sequential execution. Note that similarly to Figure 6.2, although we do see improvements from having a better branch predictor, these are not as much as they were for sequential systems. This suggests that, although as we have shown in Section 6.1 branch prediction is fairly important for TLS systems, we cannot expect the same benefits in terms of performance from traditional means as we were used to have in single threaded systems.

6.3 How Hard is Branch Prediction for TLS

The predictability of a stream of branches can be estimated by how compressible this stream is. Using the methodology proposed in [50] we compute the number of “bits” of information conveyed per branch. A larger number of “bits” indicates a stream with more entropy and that is, thus, likely more difficult to predict. Figure 6.4 shows the number of bits per branch for the sequential execution and for TLS executions with 2, 4, and 8 processors. We can see in this figure that in all cases the number of “bits” per

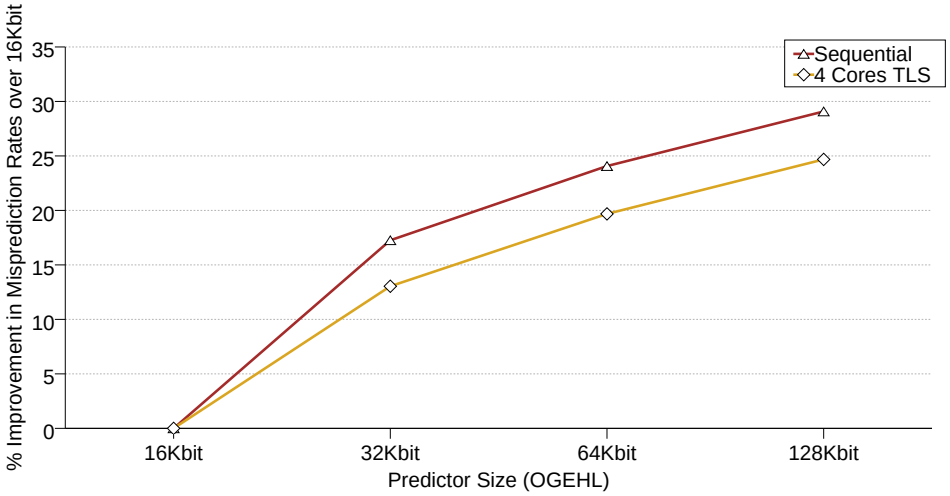

Figure 6.3: Misprediction rates for TLS execution on 4 processors and sequential execution for varying predictor size.

branch is larger with TLS than with sequential execution and, for TLS, increases with the number of processors. Note also, that the differences here in intrinsic predictability may or may not lead to similar differences in branch prediction accuracy, as the latter depends on how well a predictor handles the outcome stream.

In fact branch prediction under TLS execution has to deal with new, previously non-existing, issues which degrade the performance of branch predictors. First of all, under TLS the branch history is partitioned among the cores, such that no predictor has access to the full branch history (a point that also suggests that predictors relying on large history registers will suffer the most). Additionally, TLS threads may be squashed and re-executed multiple times – possibly on different processors. Moreover, threads are dynamically scheduled on processors and often out-of-order (predictors see random parts of the branch history). We therefore believe that in order to bridge this inherent gap in predictability, TLS systems should be combined with an execution model able to deal with hard-to-predict branches (Chapter 7). An additional side-effect of TLS execution is that prediction outcomes (i.e., correct prediction or misprediction) in re-executed threads distort the measurements of accuracy of predictors. The problem is that the same *dynamic* branch instance may occur multiple times if a thread is

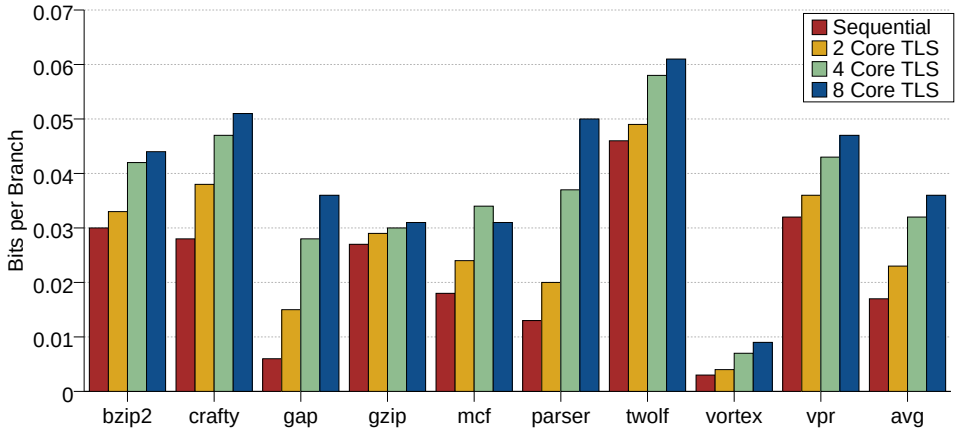

Figure 6.4: Branch entropy: number of “bits” of information conveyed per branch for different number of processors.

squashed and re-executed, while it should have been executed only once in a sequential or non-speculative execution. For this reason, we advocate that when comparing branch prediction strategies for TLS, the misprediction rates be reported for branches in the committed threads only [1]. For the rest of the thesis the reported misprediction rates are only for the committed threads.

[1] Note that the same applies for some other metrics, such as cache miss rates.

Chapter 7

Combining TLS and Multi-Path Execution

7.1 Basic Idea

In the previous Chapter we showed that branch prediction is inherently more difficult under TLS execution but at the same time more important in terms of performance. We also showed that resorting to traditional ways of improving the branch prediction mechanisms are not as effective as they are for single threaded execution. An alternative way is to use conventional branch prediction for most of the branches and employ Multi-Path for the hard-to-predict ones.

Unfortunately, MP has only been studied for fairly wide single core out-of-order systems, where it was reported to be able to save many pipeline flushes, albeit at a high cost in additional instructions executed. For TLS, however, which relies on simpler cores, this is not the case. As we will show in the next sections, MP is able to dramatically reduce the pipeline flushes due to branch mispredictions, at a reasonable cost in additional instructions. Multi-Path execution in fact makes more sense for smaller cores, like the ones we use in our multi-core, than it does for larger ones. Multi-Path execution is often considered a rather wasteful approach to remove costly branch mispredictions, since we typically have to execute a significant amount of extra instructions. For our simpler cores, however, due to their shallow pipelines, this is not

the case since branches are resolved relatively fast. Small resolution times also suggest that one should be able to employ MP execution to a higher number of branches, since it is less probable that we will encounter too many branch mispredictions before we resolve the previous ones. Note that although for sequential execution, smaller resolution times result in reduced performance benefits, this is not the case for TLS, where mispredictions cost more than they do for sequential execution. Of course, in order to be able to support MP execution efficiently, we need to enhance our cores with multi-threaded execution. The reason for this is that under Multi-Path execution we have to be able to perform fast register file copying, when we decide to follow alternate paths. We will further discuss how this affects our system in a subsequent section.

Figure 7.1 depicts our proposed combined execution model. Under this scheme we have two operational modes, normal TLS and MP mode (we only show four paths for clarity). We enter MP mode when we find a hard-to-predict branch and if there is a free context available on the core running the thread. When the branch is resolved, we follow the correct path and discard the incorrect one. When all pending hard-to-predict branches are resolved we exit MP mode. In the next Section we describe the necessary hardware support as well as how our scheme operates in more detail.

7.2 Extending the TLS Protocol to Accommodate MP

Supporting Multi-Path execution in a TLS system is somewhat different from what it is for single threaded systems. More specifically, these wrong path threads have to somehow be accommodated within the speculative multithreading protocol, so that they are both fed the correct memory values (to the extend this is possible) and also do not create unnecessary invalidations (for the TLS threads). Additionally, we do not want threads executing on the wrong path to trigger the squashing of subsequent threads, which we wouldn't otherwise squash. In order to deal with these issues, we have to slightly modify the existing TLS protocol, so that it can differentiate between the normal TLS threads and the threads that may be executing on the wrong path. Note that combining TLS with MP execution is fairly different from the combined execution model proposed in Chapter 4. The reason is that under the combined TLS/MP

execution model, one of the additionally created threads will have to commit its state (whereas the HT/RA threads never actually committed their state). We thus have to make sure that wrong propagation of values to the threads that follow alternate paths never happens.

7.2.1 Additional Hardware

In order to support this operation we need to keep some extra information which has to do with the path each of the threads in MP mode is following and how many outstanding paths we currently have. We thus, hold per thread an extra bit, the *MP* bit, which indicates whether the thread runs on MP mode or not. To keep track of the outstanding paths we require per task an additional three bit counter (we allow six paths). We call the bits of this counter the *PATHS* bits, and we increment them every time we create a MP thread and we decrement each time we kill a MP thread. So far, the two threads that follow alternate paths are identical. In order to be able to differentiate between them, we keep per task an additional three bits, the *DIR* bits, which indicate the direction that the thread has followed since it started in MP mode(i.e., taken or not-taken). Since while executing in MP mode, we are not sure which of the two threads will be the one that will be kept and which will be discarded, we have to treat them as if they were the same thread in terms of the version of data they read. At the same time we must be able to differentiate which exposed reads have been performed by which thread so that we do not unnecessarily kill any thread because its wrong path clone performed a violating access. Since we keep information about exposed reads at the cache lines, we have to augment them with the DIR bits as well. When two threads that follow alternate paths perform a read, they do not need to consult their DIR bits (i.e., they only use the version ID as normal TLS threads). However, when they keep their own copy of the cache line, instead of only tagging it with the version ID, they also use the DIR bits. In this way when a read is found to be violating, we can check whether it was performed on the wrong or the correct path, and restart only if necessary. When a subsequent thread wishes to read a value, it may read it without consulting these extra bits. Note that because under MP mode all the stores are kept in the store buffers until the branch commits, control-speculative values are not propagated to the versioned memory and

neither are, thus, transparent to the TLS protocol. Thus, threads that run on MP mode, cannot cause squashes to other threads. When a branch that triggered MP execution is resolved, the corresponding PATHS bit is decreased by one, and the DIR bits of the two threads are checked accordingly. Finally, we need an additional three-bit counter, whose bits we term the *CUR* bits, which denote which is the most recent outstanding branch that caused the triggering of the MP mechanism (MP branch). When such a branch is resolved the counter is incremented by one. When the MP bit is set to zero (no more paths), we reset the CUR bits as well.

If a predecessor thread performs a store that violates a load in a thread running on MP-mode, we do not restart the thread immediately. We instead wait until the all the branches are resolved and we take action then accordingly (in a fashion similar to delayed disambiguation schemes).

As Figure 9.1(a) shows, normal TLS threads have zeroes to all these extra bits. When a low confidence branch is encountered (Figure 9.1(b)), the thread executing follows the predicted path, while another thread is spawned that follows the alternate path. Both threads set their MP bit to 1, indicating they run in MP mode, update the PATHS counter and consult it so as to set the DIR bit accordingly (if PATHS is 1, the first DIR bit should be changed). When a second low confidence branch is encountered (Figure 9.1(c)), the PATH counter is updated (the MP is already set for the two pre-existing threads so it only needs to be set on the newly created thread). Note that now the spawnee thread cannot follow any path as it would have to update all of its cache lines, it thus follows the path that will leave its DIR bits the same (in this case the ’000’ path). Once the first branch is resolved, the thread that was executing on the wrong path is discarded and the remaining threads decrement their PATHS counter by one. Once the PATHS counter is zero again, it means that the TLS thread now operates in normal TLS mode.

Two implications have to be dealt with: the first is that we have so far silently assumed that the branches will be resolved in the order they created the paths. However, since this may not be true we must have a way to either prevent it from happening or keep track of which branch causes the creation of which thread. We choose the first, since the ROB already provides the required support. More specifically, we do not

inform the system that the MP branches are resolved if all the previous MP branches in the ROB have not yet been resolved. As such we can be sure that the correct ordering is maintained. The second implication is how do we handle spawn instructions while in MP mode (so that we don't create threads on the wrong path). In order to prevent the spawning of threads while on the wrong path, we pessimistically suppress them, in our effort to make the MP-execution as transparent as possible to the normal TLS execution.

Of course in order to create the thread running on the other path, we need fast register file copying (or fast copying of the rename tables). For this reason we rely on cores with multi-threaded execution support [78]. MP threads will have to be created and mapped on the same core. Note that live-in passing from spawnee to spawned thread for these threads is not performed via memory as it is the case for normal TLS threads, since the compiler is not aware of their existence.

7.2.2 Mapping TLS Threads

Supporting multiple contexts on the same core for the purposes of MP execution provides additional mapping options for normal TLS threads as well. If we map TLS threads on the same core but on a different context, a policy we call *SMTFirst*, we will have faster thread spawns and commits. Unfortunately by doing so, we can no longer use the contexts of the core to perform MP execution. Additionally, because TLS threads are by construction similar [1], they will probably contend for the same resources and slow each other down. An alternative approach is to first try to map TLS threads to free cores and use the contexts only if at a given execution point we have more threads than cores. In this way we both help TLS and also give our system more opportunities to exploit MP execution. Because the time spent in MP mode is far less than that spent executing normal TLS threads, we manage to minimize the contention we have per core. We call this mapping policy *CMPFirst* since it gives priority to empty cores for TLS threads. In Section 8.10 we compare the two policies and show that the *CMPFirst* is the most efficient mapping policy both for TLS and for our combined scheme.

[1] As we mentioned in Section 2.1.2 they are often created from iterations of loops.

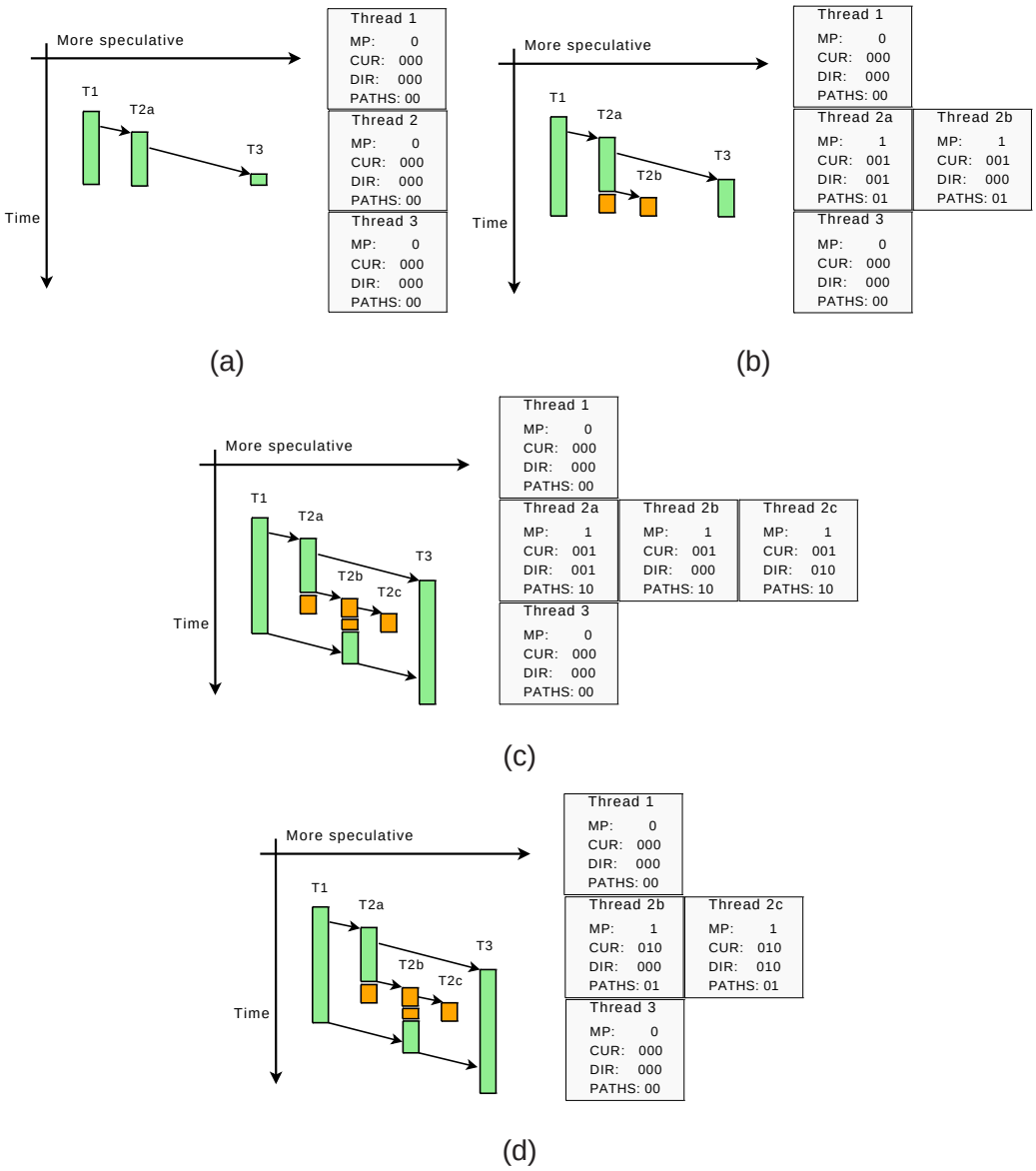

Figure 7.1: Combined TLS/MP Scheme along with the additional MP, DIR, CUR and PATH bits: (a) Normal TLS spawns threads when it encounters spawn instructions. (b) On a ”hard-to-predict” branch in T2a we create a clone of the thread that follows the alternate path. MP and PATH bits are set to one and DIR bits are set accordingly and the CUR bits are set to one. (c) A second ”hard-to-predict” branch is encountered in T2b. The MP bit of the new thread is set to one, the PATH bits are incremented and the DIR bits are set so that they do not change the DIR bits of the spawnee thread. The CUR bits are left as they were. (d) The first ”hard-to-predict” branch in T2a is resolved, we discard the thread that was on the wrong path (T2a) and continue execution. We decrement the PATHS counter and increment the CUR bits.

Chapter 8

Analysis of the Combined TLS/MP Scheme

8.1 Additional Hardware to Support MP

In addition to the base architecture we require support for multiple contexts, a confidence estimator and some extra bits in the cache. The confidence estimator, is an important mechanism for MP execution, because it is used to trigger the MP execution mode. We use a 24-Kbit enhanced JRS confidence estimator [31] which uses 11-bits of misprediction history. Using CACTI we found that the overhead of the extra L1 bits needed to store the additional information for keeping track of whether we are on MP mode and which direction we are following was small enough, that it did not affect the number of cycles to access it. In order to allow Multithread execution within the cores all structures were shared among the different contexts. A round robin policy has been used to select which thread we should fetch from. The main microarchitectural features are listed in Table 8.1. The top part lists the baseline parameters as shown in Table 3.1, and the bottom part lists the additional parameters.

Table 8.1: Architectural parameters with additional parameters required for MP support.

Parameter	TLS (4 cores)
Fetch/Issue/Retire Width	4, 4, 4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80, 104
Ld/St Queue	54, 46
Branch Predictor	48Kbit Hybrid Bimodal-Gshare
BTB/RAS	2K entries, 2-way, 32 entries
Minimum Misprediction	12 cycles
Task Containers per Core	8
Cycles to Spawn	20
Cycles from Violation to Kill/Restart	20

Extra Hardware per Core	
Additional Contexts	3
Confidence Estimator	8K Entries / 3bits JSR

8.2 Performance of the Combined TLS and MP Execution Model

Figure 8.1 depicts the performance of MP execution, TLS execution, a branch prediction enhanced TLS execution, and our combined scheme. The enhanced TLS system has a branch predictor of double size. Note that MP execution can only get significant benefits over sequential execution for *gzip* (9.3%), *mcf* (15.2%) and *vpr* (23.4%). Note also that for *gap* and *vortex* it is actually slower than the sequential system. This stems from the inability of the confidence estimator to accurately find branches that would have been mispredicted and it thus unnecessarily increases the contention for functional units in the core. Despite the slowdown in these three applications, MP execution still manages to improve performance by 5.4% on average, a result which is in line with those previously reported in [42].

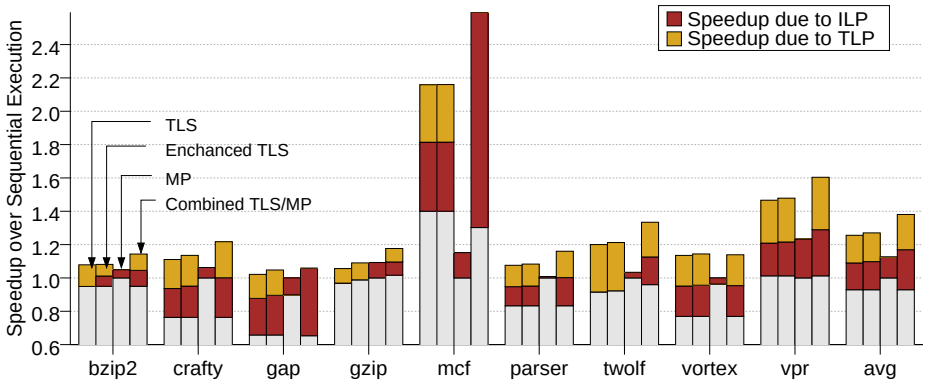

Figure 8.1: Speedup of 4 core TLS, a TLS system enhanced with a predictor of double the size, MP execution, and our combined scheme over sequential execution.

For the base TLS system, the performance is always substantially better than that of sequential execution, as it was shown in previous sections as well. In the same graph we also see an enhanced TLS scheme, which uses a branch predictor with double size. As expected from the previous analysis (Section 6.2), it is not significantly better than the base TLS.

The combined scheme is able to perform significantly better than both MP execu-

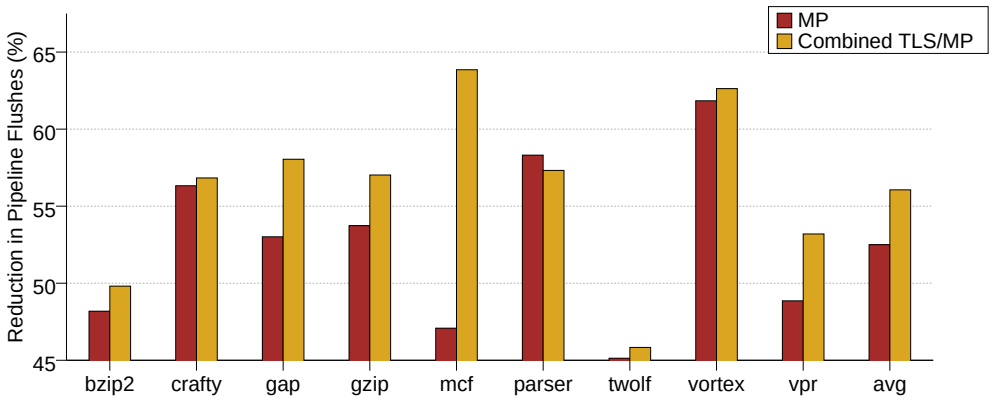

Figure 8.2: Reduction in pipeline flushes for MP (over sequential) and the combined TLS/MP scheme (over the baseline TLS).

tion and the base TLS. More specifically, the combined scheme improves performance over TLS by as much as 20.1% (*mcf*), with an average of 8.8%. The only case where our combined scheme does not get any improvement over TLS is *vortex*. However, the misprediction rate for *vortex* is below 1% and as such there is little room for improvement. Perhaps the most interesting results are that of *gap* and *mcf*, which although they lose all of their TLP, they are still able to achieve speedups over TLS. In fact for *mcf* due to its parallel execution, it even manages to get a TLP slowdown (the grey shade is lower than it is for TLS), because tasks executed in parallel are forced to wait for a long ammount of time before they get squashed. However, these squashed tasks are still useful since they prefetch for the main thread. When we compare our scheme over the MP execution, the achieved speedups are even more pronounced (29% on average). Overall the combined execution model seems to enjoy an additive effect in terms of performance and manages to gain speedups over the sequential execution both due to benefits arising from improved ILP and extracted TLP.

In Figure 8.2 we depict the reduction in pipeline flushes that the baseline MP and our combined scheme are able to achieve compared to sequential execution. It is interesting to note that our combined scheme is able to reduce significantly more pipeline flushes than the MP system can. The reason for this is that under our scheme, we are able to perform multiple MP executions across cores, and thus save more branch mis-

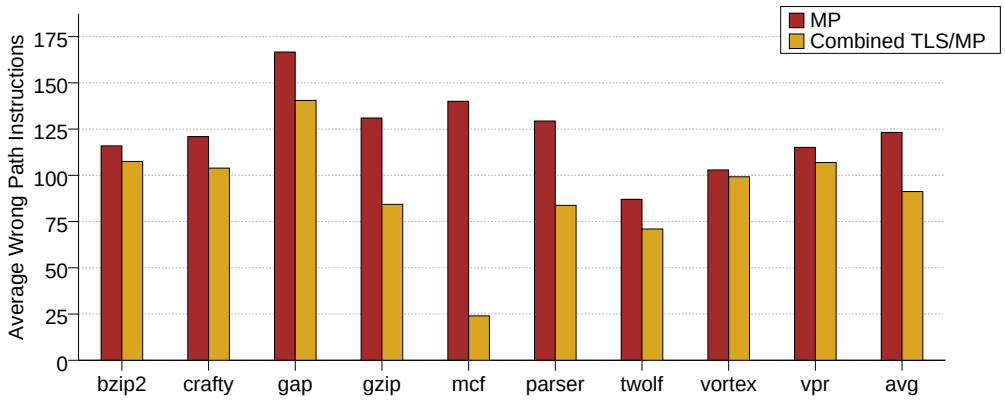

Figure 8.3: Average number of instructions executed on the wrong path for MP and the combined TLS/MP scheme.

predictions. The only benchmark where MP does better than our combined scheme is *parser*, where the confidence estimator of the combined scheme fails to correctly pinpoint the hard-to-predict branches with the same accuracy it does for the MP case. As it is to be expected, the increased coverage that the combined TLS/MP scheme enjoys, comes at an additional cost in the number of instructions executed on the wrong path over the MP scheme. What is perhaps more interesting to note is that the average number of instructions executed on the wrong path increases as well for the combined scheme. As Figure 8.3 shows, the increase is more significant for the memory bound applications *gap*, *mcf* and *parser*. The reason for this is that under the combined scheme we employ MP more times and thus increase the probability that we will perform MP on a branch dependent on an access to the memory. This means that for the combined scheme the branch resolution time is larger than it is for the MP scheme. Increased resolution time in turn, results in more time to execute instructions on the wrong path. Of course in case these branches would have been mispredicted, this also means that we were able to save a costly miss event.

8.3 Sensitivity to Microarchitecture Parameters

8.3.1 Better Confidence Estimation

In Figure 8.4 we can see what the effect is of using a better confidence estimator. The results suggest that the 48K bit confidence estimator is able to improve the results only marginally, from the 24K bit one used throughout this Chapter. More specifically the fourfold increase in size translates to only 1% improvement on average in the overall execution time. Of course instead of using a small gshare-like structure to perform the confidence estimation one could use a perceptron-based one [8] or one coupled with a value predictor as in [5], which would perform much better. As the same graph shows, being able to perform better estimation of the hard to predict branches can lead to an up to 12.1% performance improvement on average over our scheme. Note that the extent to which better confidence estimation can lead to improved performance is also conditioned to the system's load and to how close mispredicted branches are.

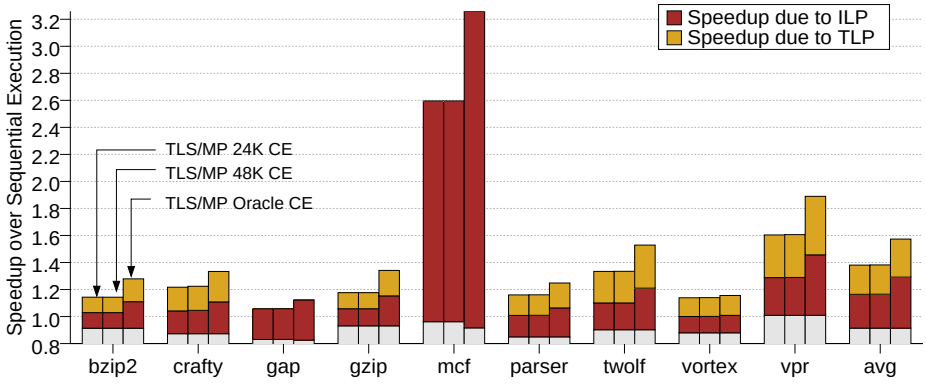

Figure 8.4: Speedup of the proposed combined TLS/MP scheme over sequential execution for different confidence estimator sizes, and oracle confidence estimation.

Figure 8.5, shows that the additional speedup comes from a significant increase in reduction of the pipeline flushes. In fact the oracle scheme is able to remove almost 90% of the pipeline flushes. The remaining mispredictions in fact, correspond to branches that have a higher degree of clustering than the allowed concurrent paths and thus cannot be removed. Note that the achieved speedup is not that of removing 90%

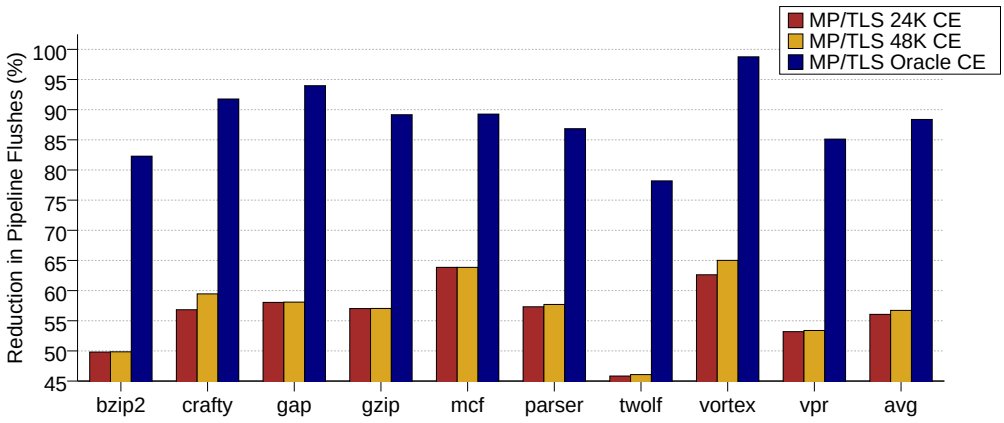

Figure 8.5: Reduction in pipeline flushes for the combined TLS/MP schemes with the base 24Kbit confidence estimator, a 48Kbit, and an oracle one.

of the mispredictions, since in this case we execute more instructions which actually content for the common resources. Additionally, the delayed disambiguation that we are forced to employ leads to a further slowdown. The average number of instructions executed on the wrong path on the other hand, does not seem to change significantly (Figure 8.6).

8.3.2 Limiting the Available Paths

By limiting the number of paths we follow, we may still be able to achieve some speedups, albeit smaller. Figure 8.7, shows the bottom line speedups achieved over sequential execution for our combined scheme when we are only allowed to follow two paths (Dual-Path) and when we follow four paths. The most significant speedup is achieved for *mcf* (10.4%), where the combined TLS/MP is 3.8% better than the combined TLS/DP scheme. This speedup is not achieved because the MP scheme is able to deal with clustered branches but because it is able to reduce the importance of the confidence estimator (as mispredictions of the confidence estimator do not remove opportunities for branch removal). Figure 8.8 shows the corresponding reduction in pipeline flushes. An important thing to notice is that this result suggests that if we were able to built a more accurate confidence estimator, we may provide similar results with

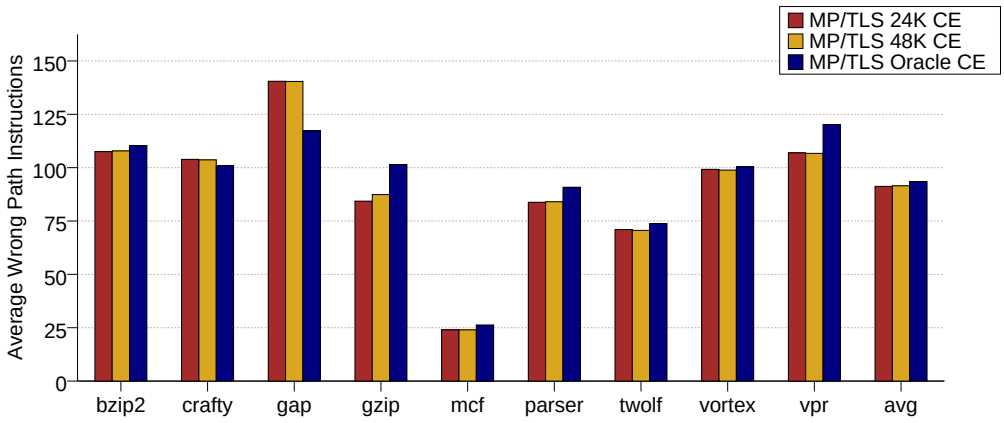

Figure 8.6: Average number of instructions executed on the wrong path, for the combined TLS/MP schemes with the base 24Kbit confidence estimator, a 48Kbit, and an oracle one.

fewer contexts. This in turn means similar results with a smaller amount of wrong path instructions. Figure 8.9 shows that the average number of instructions is also reduced significantly.

8.3.3 Impact of Mapping Policy

In Section 7 we noted that correctly mapping the TLS threads is likely beneficial both for TLS and for our scheme. For TLS the argument for the *CMPFirst* policy is that threads are by construction similar, and as such they contend for the same resources. At the same time mapping TLS threads on different cores allows us to perform MP execution. As Figure 8.10 shows, there is a huge improvement when we prioritize the mapping of newly created threads to empty cores. More specifically, *gap* improves by 32 % and *mcf* by 28%, while the average improvement is 19.2% over SMTFirst. Note that for most of the applications the *SMTFirst* policy loses all the speedup contributed by prefetching (ILP part) and is able to achieve speedups only due to TLP.

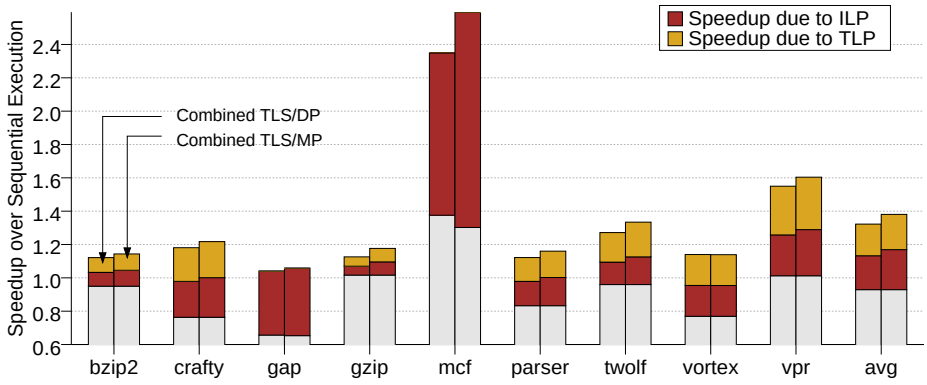

Figure 8.7: Speedup of the combined TLS/DP and the combined TLS/MP schemes over sequential execution.

8.3.4 Using a Better Branch Predictor

We finally performed an analysis of how the proposed scheme would perform with a better branch predictor, namely the highly accurate OGEHL [66]. For this study we replaced our hybrid predictor with a 32Kbit OGEHL for both our scheme and the baseline TLS and sequential executions. Figure 8.11 shows the speedups achieved by the combined scheme when using the simpler hybrid predictor and when using the complex OGEHL. Interestingly enough although the results are fairly similar, the hybrid scheme works better, although both of them outperform the baseline TLS.

As Figure 8.12 reveals the main reason for that is that the confidence estimator is able to pinpoint the branches that will mispredict more accurately. At the same time the number of instructions does not change significantly (Figure 8.13), so that the overall overheads associated with the combined scheme remain the same. Of course the use of the simple confidence estimator hinders the performance of the OGEHL-based scheme, but the possibility of using simpler predictors combined with speculative multithreading techniques is quite interesting and it should be investigated further in future work.

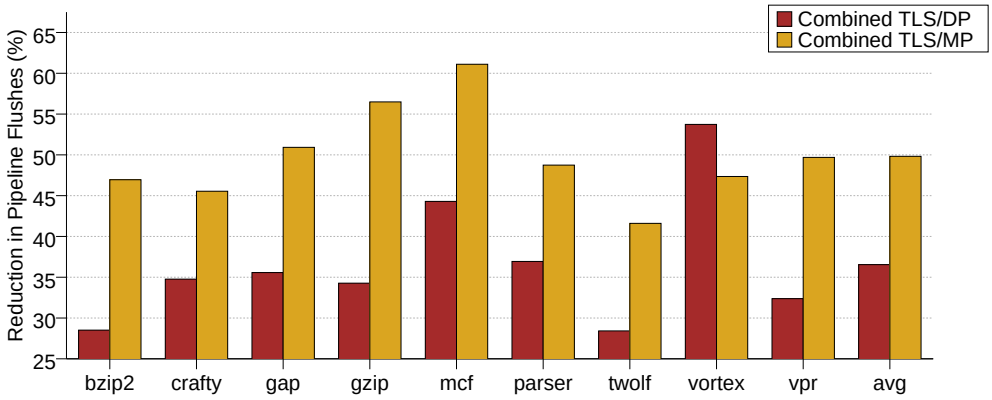

Figure 8.8: Reduction in pipeline flushes for the combined TLS/DP and the combined TLS/MP schemes.

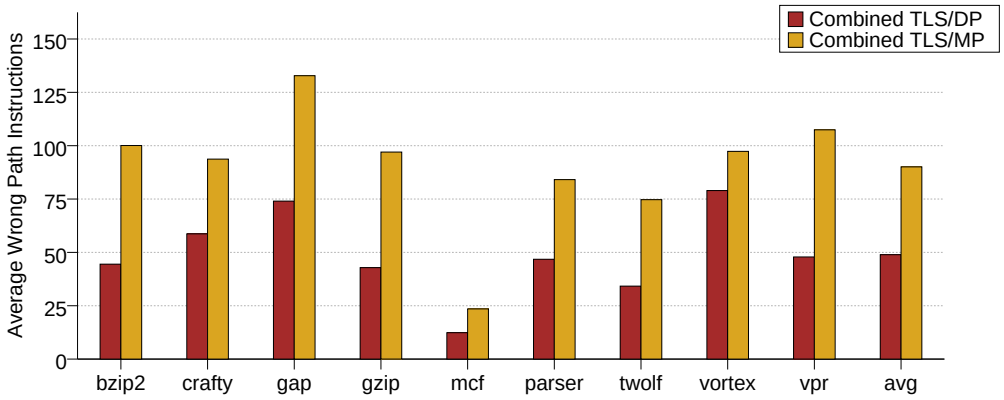

Figure 8.9: Average number of instructions executed on the wrong path for the combined TLS/DP and TLS/MP schemes.

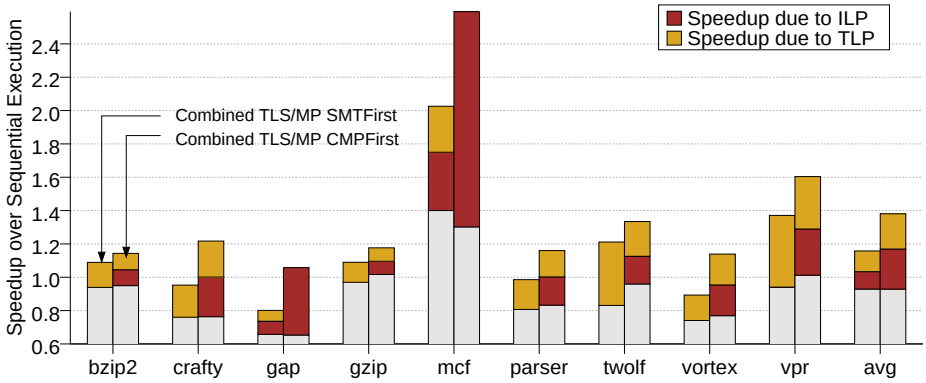

Figure 8.10: Speedup achieved by using the CMPFirst mapping policy and using the SMTFirst one for our combined scheme.

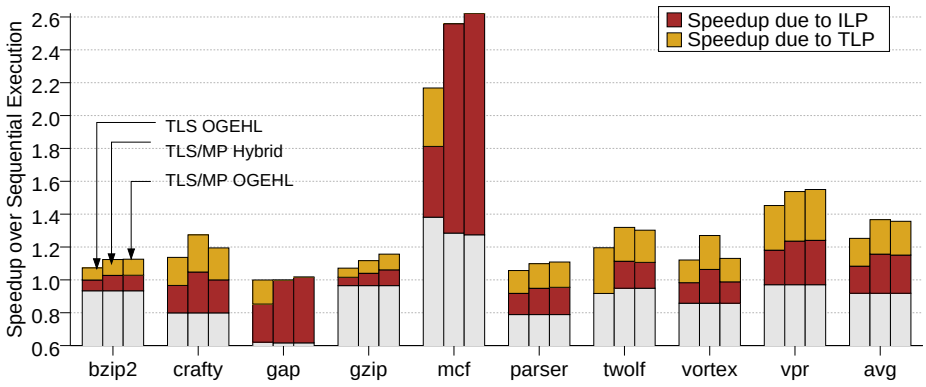

Figure 8.11: Speedup of the TLS with an OGEHL predictor, and the combined TLS/MP scheme when using the Hybrid and the OGEHL branch predictors over sequential execution (using the OGEHL as well).

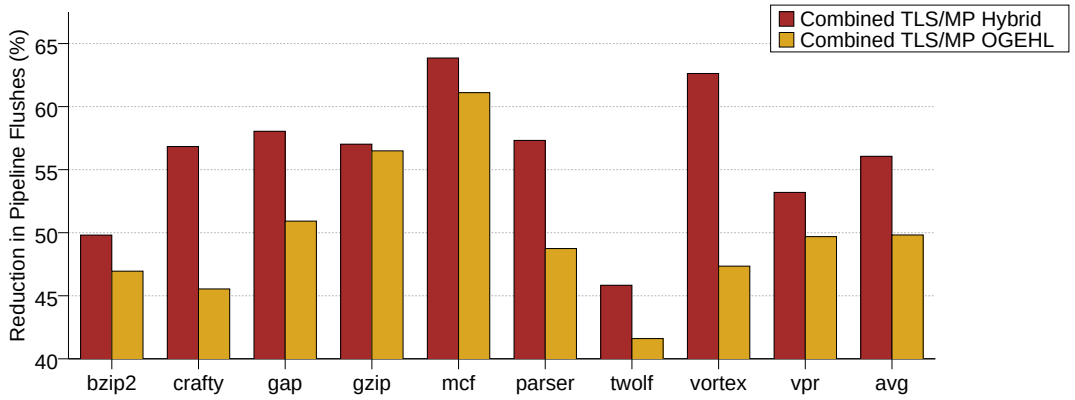

Figure 8.12: Reduction in pipeline flushes for the combined TLS/MP scheme when using the hybrid and the OGEHL branch predictors.

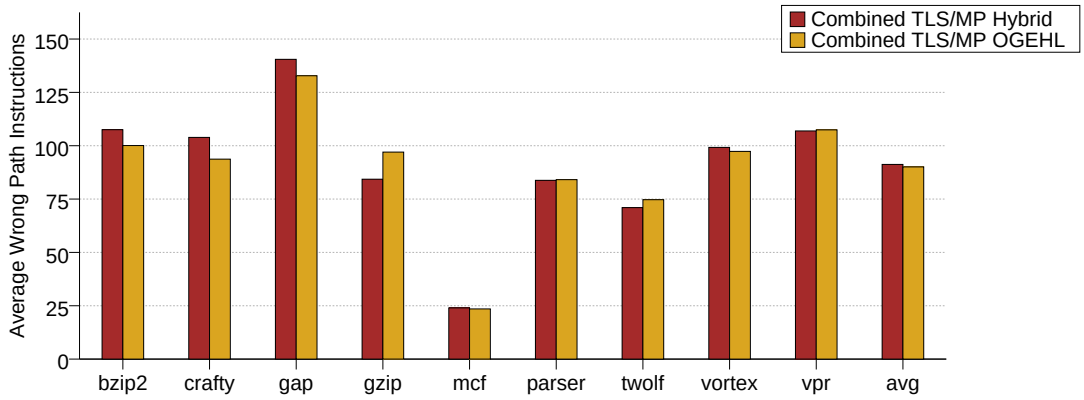

Figure 8.13: Average number of instructions executed on the wrong path for the TLS/MP scheme when using the Hybrid and the OGEHL branch predictors.

Chapter 9

Profitability-Based Power Allocation

9.1 Background on DVFS

The basic dynamic power equation: $P = CV^2Af$ clearly shows that there is a great opportunity to save power by adjusting voltage and frequency. By reducing the voltage by a small amount, we reduce power by the square of that factor. Unfortunately, reducing the operating voltage means that the transistors need more time in order to switch on and off, which also forces a reduction in the operating frequency. Dynamic Voltage and Frequency Scaling (DVFS) [51] techniques try to exploit this relationship by reducing the voltage and the clock frequency when they discern that they can do so, without experiencing a proportional reduction in performance.

Adjusting the voltage and frequency is done by means of a DC-DC converter, which changes the voltage to the desired levels. The new operating voltage is then used to drive the frequency generator, which provides the chip with the operating frequency for the corresponding voltage level. Having a means of changing the voltage and frequency, one has to decide whether to put the DC-DC converter off-chip [59, 81] or on-chip [1, 33]. Placing the converter off-chip, we are limited in that we can only change the voltage and frequency of the entire chip. A second related issue is that off-chip regulators, are generally slow. However, they consume less power and require a smaller hardware budget than their on-chip counterparts. On the other hand, this additional area and power consumption grants on-chip regulators faster response times

and allow changes of the voltage and frequency to parts of the chip.

Most modern processors have support for DVFS in order to save power or to avoid thermal emergencies [25]. Experiments done in [27] show that it is advantageous to reduce the CPU frequency for a memory intensive task, but not for a CPU-intensive task. The performance of a task with high CPU utilization is linearly dependent on frequency, and thus will suffer significant throughput loss when the frequency is lowered. A memory intensive task, however, will suffer minimal performance loss when the frequency is reduced. If a task is constantly accessing memory, then the CPU is constantly stalling and waiting for memory. Power consumption can be reduced by lowering the frequency for a memory intensive task, and system performance can be increased by running a CPU-intensive task at the highest frequency.

9.2 Basic Idea

Unfortunately, a real-life scenario for TLS systems is that a significant fraction of the threads has to be rolled-back due to dependence violations. This suggests that from a performance point of view, some of the threads are profitable, while some others are not. In fact much of the energy inefficiency of TLS stems from the fact that we spend the same amount of power to execute threads that will procure performance benefits and those that don't. In this thesis we propose to try to adapt the power consumed by threads based on their expected profitability.

We leverage the fact that modern processors, like Intel's Nehalem [24], are able to increase the operating frequency of a core if the rest of the cores are either idle or on one of the low-power modes. Instead of relying on the OS to decide how to allocate the power resources to each of the cores, we use hardware predictors to guide the power allocation at run-time. In our system, we assume four power modes: the *very-low-power* mode, the *low-power* mode, the *normal-power* mode and the *high-power* mode. Each mode corresponds to a different frequency-voltage pair. We assume that the normal-power mode is the operating mode when all of the cores are busy. We also assume that, as with Intel's Nehalem, the high power mode can't be used if all of our cores are operating at normal power mode.

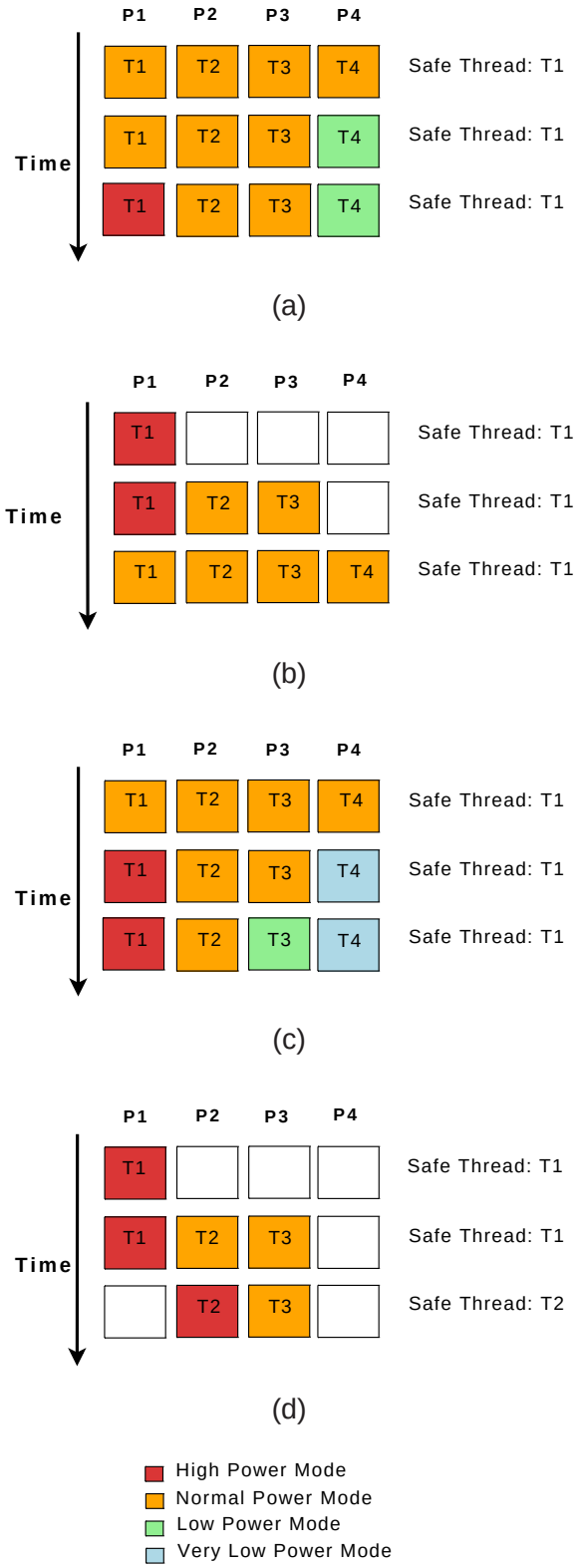

Figure 9.1: Profitability-based power allocation: (a) When all cores are occupied, only when one thread goes in low-power mode we put the safe thread in high-power mode. (b) While there are free processors (clock gated), processor *P1* which holds the safe thread *T1* is set in high power mode. (c) When a thread is predicted to squash or to be memory-bound it goes in low-power mode, when it is predicted to squash and to be memory bound it goes into very-low power mode. (d) If a safe thread (*T1*) finishes and the subsequent thread becomes safe (*T2*),

More specifically, since the only thread that will surely commit in a TLS system is the safe thread, it intuitively makes sense to try to put it in high-power mode when this is possible. As we previously mentioned, we are only able to do so only if one of the other threads is in one of the low-power modes (Figure 9.1(a)), or clock-gated (Figure 9.1(b)). On the other hand, we put threads in one of the low-power modes if we predict that they will squash due to a violation of the sequential semantics or if we estimate that they are memory bound, and as such increased frequencies/voltages will only result in minor performance gains (Figure 9.1(c)). We keep the remaining threads in the normal-power mode.

Although allocating more power to the thread that acquires the safe token is straightforward, finding the threads that will be victimized is not. One of the difficult aspects of finding the *non-profitable* threads stems from the fact that our predictions are used to guide the execution speed of the different threads and the same thread can potentially be profitable or non-profitable based on our decisions. Fortunately, by using hardware predictors like the ones presented in the following sections, our system adapts to this run-time behavior. We use two predictors: one able to predict if a thread has performed a load that it will cause it to squash, and one able to estimate if the thread is memory bound. Threads predicted to squash go into the low-power mode, while threads predicted to squash for more than three times go into the very-low-power mode. Similarly, threads estimated to be memory bound go to low-power mode.

Note that as is shown in Figure 9.1(d), when the safe token is passed to the next thread, so do the power resources. As we will show in Section 10.4, this results in more uniform distribution of the power consumed than that of a normal TLS system, and thus our scheme enjoys a better thermal behavior.

Note also that if our predictions are right and the threads do squash or are memory bound, we have saved some power. This is important since this extra power can be spent to speed-up safe threads. At the same time the extra power we consume executing safe threads in high-power mode, does not increase the average power consumed significantly. On the other hand, if our predictions are wrong, we slow down useful threads and allocate more power to non-profitable threads. Due to the central role these predictors play in our design, they have to be quite accurate. In the next two sections

we describe how they operate and discuss various design options.

9.3 Adapting to TLP

We first target threads that are not going to be useful in terms of TLP. We base our classification on a *Squash Predictor*: a dependence predictor able to predict whether a specific load will squash the thread or not. A dependence predictor has been previously proposed in [57], but it relies on global information about possibly conflicting loads and stores. Such a centralized scheme is not practical for a multi-core system like the one we use. Emulating a centralized scheme by broadcasting all the information available in a core, to all the other cores (and thus their predictors), is power inefficient and consumes valuable bandwidth. This has been previously noted in [21], where the authors designed dependence predictors for a directory-based CC-NUMA system. Instead of using a centralized scheme, they extended the directories with a few bits so as to capture the dependence behavior using only memory addresses.

We opt for a similar solution with the one presented in [21]. More specifically, we maintain a simple table of three bit saturating counters per core as is shown in Figure 9.2(a). When a speculative thread (i.e., all threads but the safe one) tries to execute a load instruction, we perform a bit-wise *XOR* of the memory address and the five least significant bits from the load's program counter and form an index. We use this index to lookup the corresponding counter from the table we mentioned before. If the value of the corresponding counter is larger than three, we then predict that the specific load is being performed prematurely and will thus cause the thread to squash, otherwise we predict that the specific load will not cause any problems. At the same time we also update the tag of the cache line that holds the specific memory address, with the five least significant bits of the program counter. The predictor is updated when we perform a store. The memory address of the store is propagated to all of the cores, since it is used by the TLS protocol to uncover any dependence violations by checking whether any of the caches holds a violating load. If the store does reveal a dependence violation, we read out the five bit field that holds the program counter of the load that last touched the cache line. These bits bit-wise *XORed* with

the memory address of the store are used to index the table of saturating counters, which we increment by two. If a thread becomes safe, and thus it cannot be squashed anymore, it updates the predictor when each of the lines that belong to it are written back. As with the previous case, we read out the PC bits and use them along with the address of the write-back request to index the counter and decrement it by one. We choose to perform this lazy update of the squash predictor, since in this way we can ensure that for each of the memory addresses for the threads that commit, we only update the predictor once (as opposed to updating on every store that does not cause a squash). This allows us to require a small number of bits per entry in the saturating counter table.

Although the main focus of this section of the thesis is not in creating novel dependence predictors, but rather on using them so as to guide power allocation, as we will show the proposed predictor is better than the previously proposed ones. Note that similarly to [21] we slightly augment the cache lines, but we only do so to hold information about the program counter that performed the specific load. As we will show in a subsequent section, this improves the prediction accuracy, since our predictor is able to disambiguate accesses to the same memory location from different sections of the code.

Having predicted which threads will get squashed, we now have to decide how much to slow them down. One option would be to stall them completely. Albeit simple, this approach can be fairly bad in terms of performance. As was pointed out in [82] even threads that do squash may be useful for prefetching reasons. By stalling threads that squash, we remove much of this desirable side-effect of TLS execution. An additional reason why this approach hurts performance is that although fairly accurate, our squash predictor may be wrong. Since the cost of being wrong is high, we would have to make our predictor fairly conservative in predicting that a thread will squash. This results of course in lost opportunity to put threads in low power mode (i.e., stall them in this case), which in turn results in not being able to put the safe thread in high power mode. We thus put these threads in low-power mode. When a thread is predicted to squash more than three times, we can be more aggressive and put the thread in very-low-power mode. Note that in this way we wrongly put a thread in very-low-power

only when we mispredict three times in a row, in one task.

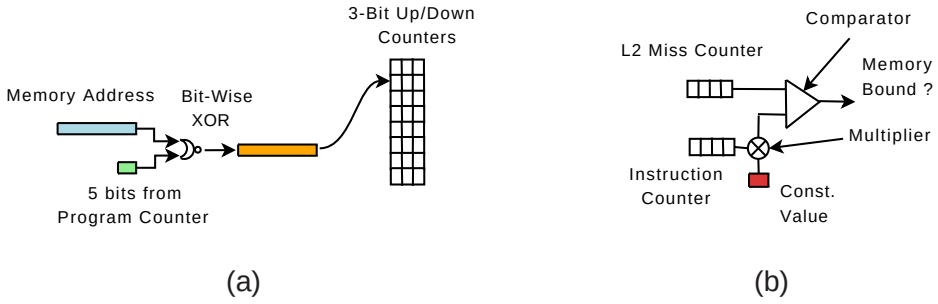

Figure 9.2: (a) Squash Predictor: The components we need are a small table of up/down counters and five bits from the PC of all loads that will be predicted. (b) Memory-Boundedness Estimator: The components we need are a level 2 cache miss counter, an instruction counter, a multiplier and a comparator.

9.4 CPI-Based Adaptation

Although by using the squash predictor we improve energy efficiency significantly, we can still do a better job by reducing the consumed power according to how much this will affect the threads performance. In fact, for memory-bound threads, most of their time is spent waiting for memory operations to be serviced. Executing them in normal-power mode is wasteful since they will consume valuable power without achieving any performance benefits out of this.

It is thus clear that we need a mechanism to decide whether a thread will wait the costly memory operations or not, since this provides an additional source of energy inefficiency. Our estimator, depicted in Figure 9.2(b), uses per core, two five bit saturating counters, a five bit multiplier, and a comparator. In order to predict whether a thread is memory bound or not, when a thread is executing we keep track of how many times we have had an unresolved memory access reaching the head of the ROB. This memory accesses are the ones that are important since they stall the pipeline. At the same time we also keep track of how many instructions we have successfully committed. When we wish to make a prediction we multiply the miss counter with the size of

the instruction window (in number of instructions) , and compare this with the number of instructions. If the product is larger than the number of instructions, we predict that we have spent most of our execution time waiting for memory and thus our thread is memory bound, if it is smaller we predict otherwise. Threads predicted to be memory bound are put in very low power mode. We consult the estimator, and thus decide on the ”memory-boundness” of the thread, each time we have a miss in the shared L2 cache.

There are two interesting points to make here. The first if that we could have done the same with the branch mispredictions. We believe that for applications with a considerable number of branch mispredictions and no significant misses in the L2 cache, such a scheme would perform better than the one presented here. The second interesting thing to note is that we make a decision of whether a thread is memory bound or not when we miss in the L2 cache. This can happen in arbitrary points in the threads execution. This also means that we may have a non-representative sample of the thread and thus make a wrong decision. To deal with this issue, we can apply a simple heuristic under which we do not allow a thread to go into very low power mode, unless we have seen a given number of instructions. In our case however, we have seen that such a heuristic is not necessary.

9.5 Applying Profitability Power Allocation to TM

In the previous sections we showed that allocating power according to profitability, provides significant benefits in terms of ED. Making speculative multithreading systems energy efficient is becoming increasingly important, especially for many-core systems like the ones we will have, according to projections made by both industry and academia. In fact better energy efficiency will allow more cores to operate at the same time and thus increase their throughput. We have shown how one can improve a state-of-art TLS system, despite the fact that it was already optimized by means of profiling (the POSH compiler used throughout this work optimizes for power as well).

Applying the proposed power allocation scheme to a TM system is quite straightforward. The only difference from a HW perspective between TM and TLS is that

under TM there is no implicit thread ordering. As such we cannot allocate more power to a specific thread based on it being safe. However under TM systems there is usually a conflict resolution policy, which is different based on the TM flavor used. We can then leverage upon this mechanism to decide which is the “safe” thread and thus guide our scheme. The squash predictor and the memory boundness estimator presented here can then be used in a similar fashion to the one described in previous chapters.

Chapter 10

Analysis of the Profitability-Based Scheme

10.1 Additional Hardware to Support Power Allocation

Each one of the cores along with its associated L1 caches form a separate voltage/frequency domain. The shared L2 cache together with the interconnection network belong to a different domain as well (which is fixed). On-chip regulators are placed per core so as to implement the different power domains, in a similar fashion to [41]. In order to synchronize communication between the distinct domains that operate asynchronously to each other we use the mixed-clock FIFO design proposed in [18].

We only assume four voltage and frequency domains, as is shown in Table 10.1, similarly to the offered domains in current commercial designs (i.e., the Super Low Frequency Mode, Low Frequency Mode, Normal Frequency Mode and High Frequency Mode used in [25]). All cores operate at the normal power mode except if our predictions dictate we should do otherwise. The cost for changing a power mode depends on the voltage swing and it is modeled to be 1 ns per 10mV in accordance with [41].

Our scheme requires on top of the baseline TLS support, the necessary hardware to perform the squash prediction and the CPI estimation (per core). More specifically, for the squash predictor we need to augment the tags of the cache lines to hold the

five least significant bits of the program counter performing the access. Using CACTI we found that the overhead of these extra bits was small enough, that as not to affect the number of cycles we need to access it. We additionally need a 5-bit bitwise *XOR* and one small table of up/down counters For the CPI estimator, we require two 5-bit counters, a 5-bit bitwise *XOR*, a 5-bit multiplier and a comparator.

Since the proposed scheme changes performance and power at the same time, we need to use a combined metric so as to be able to quantify the different design points. One such metric proposed in [12] is the Energy Delay product (ED), which allows us to quantify both power and execution time at the same time. Since the energy component of ED is already using the execution time (i.e., Energy = Delay x Power), the metric emphasizes more on execution time than it does on power. Other metrics like ED^2 or ED^3 emphasize even more on delay. Since we feel TLS and any speculative multithreading technique aims at reducing execution time, we believe that emphasizing more on execution time than on power is the correct thing to to do. At the same time putting too much emphasis on the execution time component, may make any power savings/losses negligible. We thus feel that ED is the correct metric to use to evaluate our technique. All the thermal simulations are performed using Hotspot [67].

10.2 Comparing the Profitability-Based Scheme with Static Schemes

Figure 10.1 depicts the bottom line results when we compare our scheme with the three static power modes, namely the *very-low-power*, the *low-power* and the *medium-power* modes. Note that we do not compare against the *high-power* mode since we assume that having all the cores operating in the high-power mode is not allowed due to physical constraints.

As Figure 10.1 shows, the normal-power mode is better than both the very-low-power mode and the low-power mode, because although both of them consume considerably less power, they are too slow. Interestingly enough for the memory-bound *Mcf* the low-power more is better than all the other schemes, since for that frequency the threads execute in a way that reduces the number of squashes significantly (val-

Table 10.1: Architectural parameters used along with extra hardware required for the proposed scheme and the different power modes available in the simulated system.

Parameter	**TLS (4 cores)**
Fetch/Issue/Retire Width	4, 4, 4
L1 ICache	16KB, 2-way, 2 cycles
L1 DCache	16KB, 4-way, 3 cycles
L2 Cache	1MB, 8-way, 10 cycles
L2 MSHR	32 entries
Main Memory	500 cycles
I-Window/ROB	80, 104
Ld/St Queue	54, 46
Branch Predictor	48Kbit Hybrid Bimodal-Gshare
BTB/RAS	2K entries, 2-way, 32 entries
Minimum Misprediction	12 cycles
Task Containers per Core	8
Cycles to Spawn	20
Cycles from Violation to Kill/Restart	20

Extra Hardware per Core	
Squash Predictor	2K Entries / 3bit
Instruction Counter	5bits
L2 Miss Counter	5bits
Bitwise XOR	10 x 2-input XOR gates
Comparator	1

Power Modes	
High Power Mode	5.0GHz / 1000mV
Normal Power Mode	4.0GHz / 950mV
Low Power Mode	3.0GHz / 900mV
Very Low Power Mode	1.0GHz / 700mV

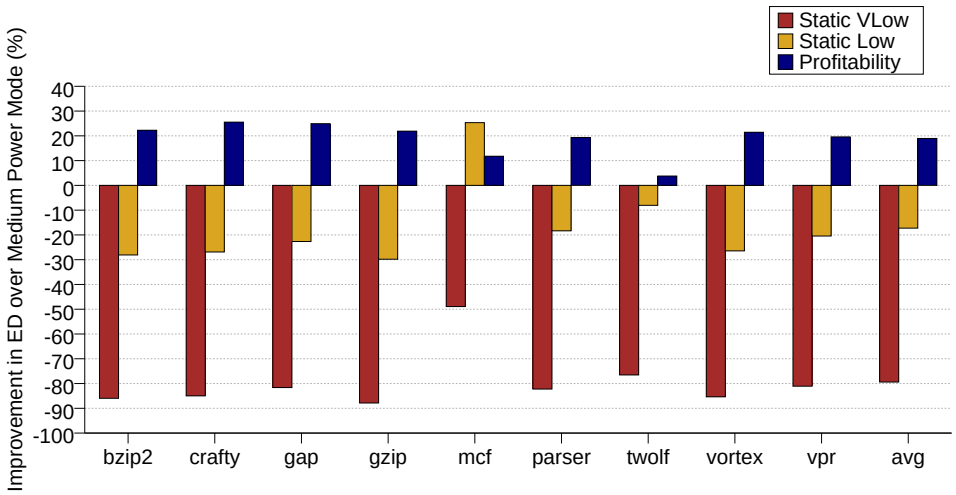

Figure 10.1: Improvement in ED over normal-power mode for very-low-power mode, low-power mode and the profitability-based scheme (%).

ues are consumed right after they are produced). We see that our scheme is able to provide a better trade-off, and it is thus 18.9% better on average than the best static scheme. Note that for some benchmarks, like *Crafty* and *Gap*, we are able to improve ED by 25.5% and 24.9% over the normal-power one. The reason for this is that for this benchmarks our squash prediction scheme works really well. In the next sections, we provide a detailed analysis of how our system is able to achieve these significant benefits in ED.

10.3 Performance-Power Analysis

Figure 10.2 depicts the speedup (or slowdown) of the static power schemes and our profitability-based scheme over the normal-power mode. Note that frequency changes are detrimental in terms of performance. In fact the very-low power mode is almost 67% slower than the base frequency system, whereas the low power mode is 16% slower than it. On the other hand the performance of the profitability-based scheme is always better than that of the base operating frequency. On average the profitability scheme is 14% faster, with *Bzip2, Crafty, Gzip* and *Vortex* achieving speedups close to

20%.

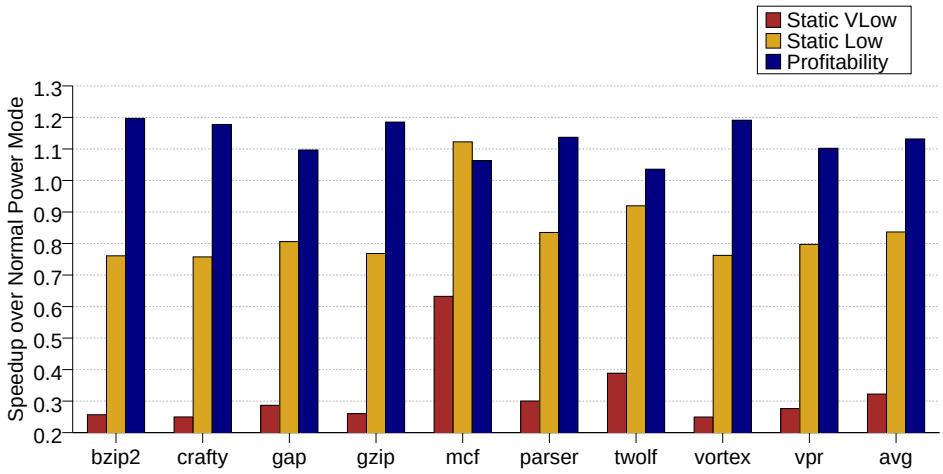

Figure 10.2: Comparing the two static power schemes, the very-low-power and the low-power modes, and our profitability-based one in terms of speedup (Normalized over normal-power mode).

At the same time, the power consumed is 7.4% on average more than that of the normal-power mode scheme. Note that the other static schemes are able to save far more power (49% for the very-low power mode and 16% for the low-power mode on average). However as we showed in the previous graph this comes at a fairly large cost in terms of performance. We thus believe that our profitability based scheme is a far more reasonable approach in terms of energy efficiency than any of the static schemes.

10.4 Thermal Analysis

Figure **??** depicts the transient thermal behavior for the base TLS system operating in normal-power mode and the profitability-based one for *Parser*. As the figures reveal, *core0* consumes most of the power and as such has the highest average temperature. Note that even though the profitability-based scheme consumes more power on average than the base TLS, the transient behavior is much better. This results in a significant reduction in the temperatures observed, while it is also interesting that the thermal gap

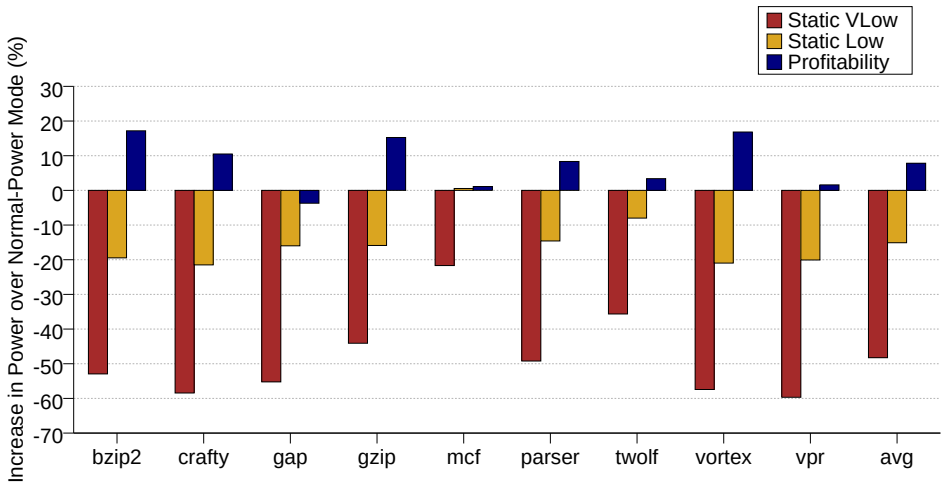

Figure 10.3: Comparing the two static power schemes, the very-low-power and the low-power modes, and our profitability-based one in terms of power (Normalized over normal-power mode).

between the processors becomes smaller as well. We believe that these two figures present a strong motivation in employing the proposed form of adaptivity. The results for the rest of the applications exhibit similar, if not better behavior.

10.5 Effectiveness of the Squash Predictor

Figures **??** shows how our scheme works when it is guided only by the Squash Predictor only. We only use the squash predictor so as to compare them without interference from the memory boundness estimator. The left bar in each graph shows the percentage breakdown of the power modes for a memory address only predictor, while the right bars show the same for our squash prediction scheme. In Figure 10.6 we see that for the threads that commit, the two predictors exhibit a similar behavior. However in Figure 10.7 we see that our scheme performs better than the memory address only scheme, and it is able to put more threads that will get squashed in low power mode. What is interesting to point out but it is not shown in these graphs is that when we also use the memory boundness estimator, the memory-based only scheme performs

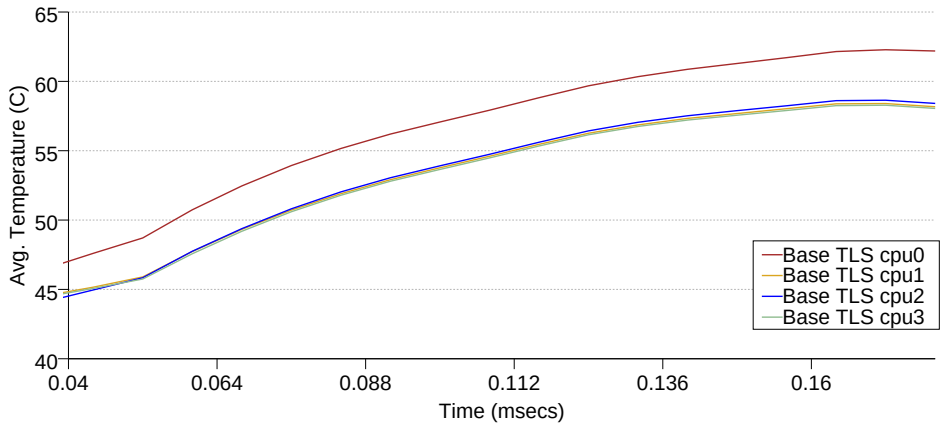

Figure 10.4: Thermal behavior per core for *Parser* for Base TLS operating at normal-power mode.

much worse then ours. The reason for this is that it is far more sensitive in the thread ordering than the proposed predictor. Of course if adding five bits for each cache line is prohibitive for a specific design, our profitability-based scheme could still perform better than any static one even with the memory address only predictor.

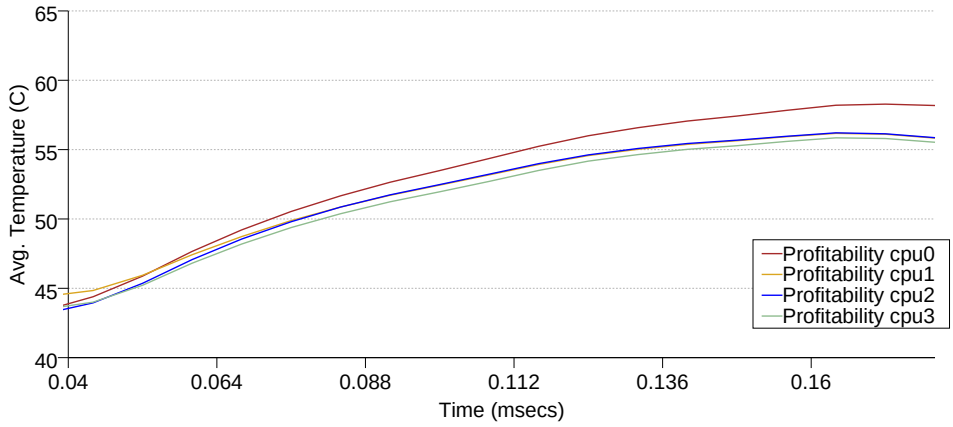

Figure 10.5: Thermal behavior per core for *Parser* for the profitability based scheme.

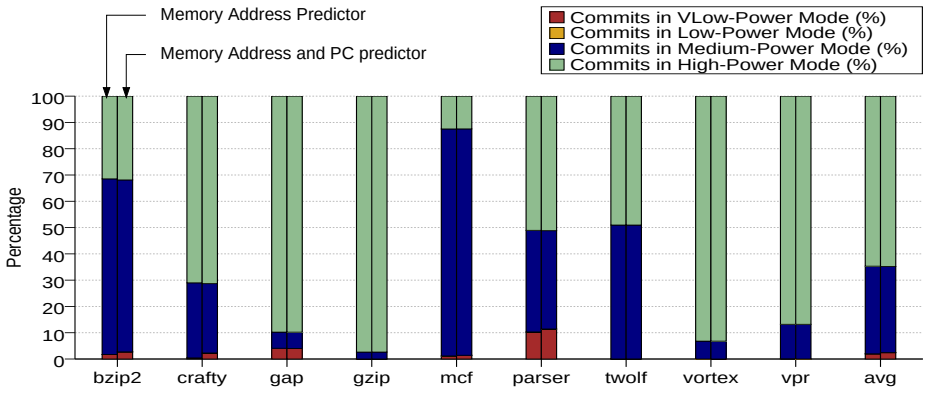

Figure 10.6: Guiding our allocation scheme using only a Squash Predictor (the memory only one and our combined) for threads that commit.

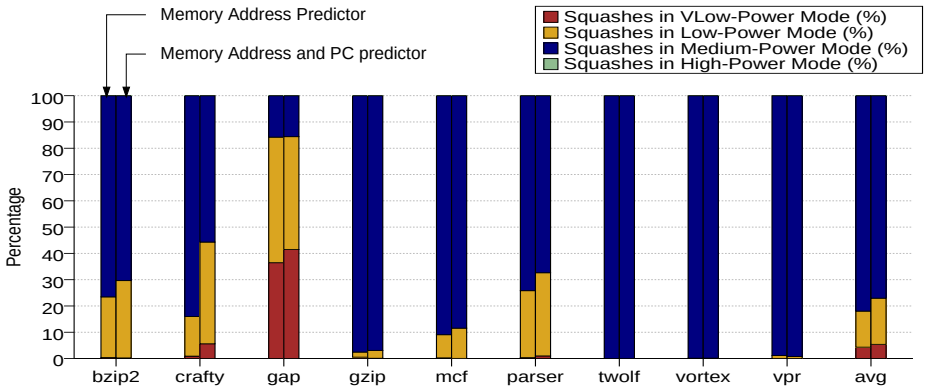

Figure 10.7: Guiding our allocation scheme using only a Squash Predictor (the memory only one and our combined) for threads that squash.

Chapter 11

Related Work

11.1 Related Work on TLS Systems

Thread level speculation has been previously proposed (e.g., [32, 45, 53, 68, 71]) as a means to provide some degree of parallelism in the presence of data dependences. The vast majority of prior work on TLS systems has focused on architectural features directly related to the TLS support, such as protocols for multi-versioned caches and data dependence violation detection. All these are orthogonal to our work. In particular, we use the system in [63] as our baseline.

11.2 Related Work on Combined Speculative Multithreading

The work in [79] showed in a limit study that it might be beneficial to continue running some threads that are predicted to squash, so as to do prefetching. However, the benefits shown there were minimal if one considers the simplifications in the simulation infrastructure used. In general, deferring the squashes so as to prefetch is not beneficial because we negate the TLP benefits. Checkpointing TLS threads was also proposed in [23]. However, checkpointing in a TLS execution model is used to improve TLP by supporting squash and rollbacks to a checkpointed state instead of to the start of the thread. In our work we employ checkpointing in a combined execution model to

allow for improved ILP while maintaining similar TLP levels. Finally, [63] also proposed a model to quantitatively break down the performance improvements between TLP and ILP. Our model is slightly more computationally intensive as it requires one extra simulation run, but our results show that it is much more accurate. Benefits due to prefetching were also reported for TLS systems in [63]. In our work we show that by employing a combined speculative multithreading approach, one can obtain further benefits from prefetching.

11.3 Related Work on Helper Threads

It has been previously proposed to use small helper threads on otherwise idle hardware resources [15, 22, 73, 85]. There, helper threads usually try to resolve highly unpredictable branches and cache misses that the main thread would have to stall upon otherwise. Because helper threads try to improve the ILP of the main thread, they fail to procure any significant benefits in applications where the out-of-order engine is able to extract most of the ILP. As we showed in previous sections, in these cases we can achieve some additional improvements by trying to extract TLP as well.

11.4 Related Work on Runahead Execution

Runahead execution [6, 28, 58] is a scheme that generates accurate data prefetches by speculatively executing past a long latency cache miss, when the processor would otherwise be stalled. Runahead execution is similar to the helper thread schemes, although instead of using different hardware resources, it uses otherwise idle processor cycles to perform prefetching. Additionally, runahead execution does not rely on the programmer to manually extract the prefetching slices. Two more recent proposals, Checkpointed Early Load Retirement [43] and CAVA [14], build on [58] by adding value prediction. In contrast with runahead execution, correctly predicting the value of a missing load eliminates the need to rollback to a checkpoint when the load returns. The work in [14] showed, however, that most of the benefits from this scheme do not come from negating the roll-backs, but rather from the fact that by value predict-

ing, prefetches are more accurate. As with the helper thread case, runahead execution tries to improve the ILP of memory bound (mainly) applications by prefetching. Our scheme is able to achieve speedups even for applications that are not memory bound. Additionally, because we are able to perform deeper prefetching than runahead execution, even for memory bound applications we are able to achieve better results.

11.5 Related work on Branch Prediction for TLS and Speculative Threads.

The work in [39] along with one presented in [10], perform an analysis of branch prediction for the Multiscalar architecture. Unfortunately, the focus of those works was very much tied to the Multiscalar architecture. Their aim was efficiently handling inter-thread branches. Intra-thread branches were not reported to be as important as the inter-thread ones, which of course was an artifact of how the threads were created for the Multiscalar processor. Inter-thread branches are only tangential to our work since in our flavor of TLS we do not have them. In fact spawning threads can only happen from control independent areas of the code. The work in [10] does enhance the intra-thread predictors by using information available in the inter-thread predictor, and is thus somewhat closer to our work, but it does not mention MP execution.

The work in [20] investigates branch prediction for execution models with short threads, which include TLS systems. It shows that branch history fragmentation can be a severe problem in such execution models and proposes a mechanism to initialize branch history. We also evaluate the effects of history fragmentation and short threads, but we additionally consider other TLS behaviors, such as re-executions and out-of-order spawns. We also implement the technique proposed in that paper and show that our proposed techniques achieve improvements that are additive to that.

The work in [54] has also briefly discussed the effects of branch prediction on TLS systems, but it does not perform a detailed study of branch prediction per se.

Other related work have dealt with branch prediction in multithreaded environments. The work in [38] targeted the Multiscalar architecture [68] and shows that having a global structure to hold history registers for all cores can lead to better predic-

tion accuracy. This approach provides similar benefits to that of history initialization in our work and in [20], but such centralized structure would not be practical in a speculative multicore environment. The work in [62] evaluates the effect that simultaneous multitreading has on the branch predictors and whether using global or local structures is better. Unlike TLS, however, simultaneous multithreading does not involve speculative thread-level execution, which leads to very different interactions between the execution model and branch prediction.

Additionally, much work has been done on helper threads to improve branch prediction (e.g., [16, 85]). In these schemes speculative threads are specifically created to run ahead of the main thread to pre-compute branch outcomes for the main thread in order to accelerate its execution. In this model the computation done in the speculative threads is discarded and no true parallel execution is achieved. This execution model is very different from that of TLS as in the latter the purpose of the speculative threads is to perform useful computation in parallel with the non-speculative thread. Since the goal in TLS is computation parallelism, the speculative threads do not directly produce branch outcomes for the non-speculative thread, although this is a potential side-effect that can be exploited, as we do in our work.

11.6 Execution Along Multiple Control Paths.

Finally, a significant amount of work has been done in the area of systems able to execute multiple control flow paths [2, 34, 44]. All these studies have shown that being able to follow multiple paths is always beneficial. In fact some of these proposals advocate following not only one but multiple control flow paths simultaneously. Recently [42] showed that combining compiler information with run-time behavior is the best approach to follow both in terms of speedup and energy efficiency. As future work we wish to explore mixing the TLS execution model with a flavor of the Diverge-Merge scheme. All these studies are based on SMT (or SMT-like) systems and assume fast register copying. The work in [19] employs a form of slipstream execution to allow multiple-path execution on highly coupled multicores. Unfortunately this proposal assumes very high coupling of the cores, through a communication queue, and it does

not scale well when the delay of the queue increases.

11.7 Related Work on Power Allocation

The work most relevant to this scheme is the one in [76]. In this work there is only one core that is fixed in high power mode and three that are fixed in low power mode. Threads are migrated to the high power core when they are predicted to be critical. Predictions are made based on a task-level criticality predictor. We showed that with much simpler predictors one can achieve significantly better results given per core voltage/frequency regulators. Recently [7] performed run-time adaptation based on criticality predictors. Although our paper shares the same ambitions, we not only cut down power but instead allocate it to the threads deemed to be profitable. An additional important issue we had to deal with, is the fact that not all our threads commit their state (in contrast to the explicitly parallel applications they used).

Fast per-core regulators like the one proposed in [80], have been demonstrated to be both fast and efficient. [41] showed that these regulators can be beneficial for fast architectural optimizations like ours. Our work assumes such regulators and it builds on top of work on synchronization among cores in different voltage/frequency isles, like the one in [18].

Chapter 12

Summary of Contributions

With the scaling of devices continuing to progress according to Moore's law and with the non-scalability of out-of-order processors, multi-core systems have become the norm. Many argue that multi-cores are here to stay, and some people are quick to argue that we will be able to continue scaling performance as we did in the past. In my opinion this will mainly depend on two things: how do we extract performance out of them, and to which extent we can scale our designs. The ambition of this thesis is to provide solutions to these questions.

While in the past more transistors would directly translate to performance, in the multi-core era this only translates to a higher number of cores. Utilizing the available hardware resources to enhance the performance of the system is thus silently alleviated from the hardware designers and placed on the compiler/programmer camp. Unfortunately parallel programming is hard and error-prone, sequential programming is still prevalent, and compilers still fail to automatically parallelize all but the most regular programs. This thesis aims at designing systems that will be able, given a sequential application to speed it up, by first speculatively parallelizing it and then enhancing the performance of the speculatively parallelized application by improving the instruction level parallelism.

The main contribution of this thesis is to enhance TLS functionality by combining it with other speculative multithreading execution models. The main idea is that TLS already requires extensive hardware support, which when slightly augmented, can accommodate other speculative multithreading techniques. Recognizing that for different

applications, or even program phases, the architectural bottlenecks may be different, it is reasonable to assume that the more versatile our system is, the better it will be able to operate.

In fact as we have shown, existing state-of-the-art TLS can be sped up significantly when it is complemented by other existing models. Thread Level Speculation, Helper Threads, Runahead and Multi-Path execution have been separately shown to improve overall performance of some sequential applications. However, given the different nature of the performance benefits provided by each model, one would expect that combining them in a combined execution model would lead to greater performance gains and over a wider range of applications compared to each model alone. Despite these opportunities no one has attempted to combine these multithreaded execution models.

The choice of execution models used for this thesis aimed at tackling two of the major ILP constraints for TLS systems, memory accesses and branch mispredictions. The first is done through the creation at runtime of helper threads, which prefetch for safer threads. The later through the use of Multi-Path execution, where on low confidence branches a separate thread is forked to follow the alternate path. Extensive experimentation showed that both these techniques are able to greatly enhance existing TLS systems, because they improve their ILP without harming the extracted speculative TLP.

Although high performance is always a goal, we can no longer pursue it blindly anymore as power consumption is a first class design constraint. In fact future multicore systems will fundamentally be limited by the on-die power density, which will limit how many cores will be operating at any given time. Thus more power efficient designs will benefit from having more cores concurrently available. Under the proposed speculative multithreaded execution models, power consumption is a major issue. This thesis showed that profitability-based power allocation schemes can improve the energy efficiency of such speculative multithreaded systems significantly. A key premise of this work stems from the idea that in a speculative multithreaded system not all threads contribute equally to performance and as such we should try to allocate more power to the more useful ones and less to the others. In fact by “stealing” power

from non-profitable threads and using it to speed up more useful ones, systems that are far more energy efficient can be designed.

Chapter 13

Concluding Remarks / Future Work

After spending four years working on speculative multithreaded systems, I think their main weakness when compared with more traditional approaches, is complexity. Correctly designing even the simplest speculative multithreaded system, is challenging (let alone verifying it). At the same time, none of the proposed speculative multithreaded systems does well for all applications, while due to speculation they spend a significant amount of power. This thesis tried to make them a more attractive solution, by showing that most of the required hardware (and hopefully complexity), is the same for most of the speculative multithreaded schemes. By combining them we can achieve significant speedups, mainly because the architecture is more versatile and can cope with all types of application behavior. In an effort to reduce the sometimes excessive power consumed, we tried to categorize threads so that we only perform deep speculation if this will translate to meaningful performance gains.

There are many interesting topics that are direct extensions of this work. The most obvious one, is to perhaps apply the same techniques to explicitly parallel applications. Even the most well behaving applications have sequential portions, which limit the potential speedups [4, 36]. Employing techniques similar to the ones proposed in this thesis, can potentially make this bottleneck a less significant one. Of course, if we consider each one of the explicitly parallel threads to be our "sequential" application, we can directly apply the proposed schemes. An interesting issue that will arise is how to manage the speculative threads created from different explicitly parallel threads, if they compete for common resources.

Another interesting extension of the work proposed in this thesis would be to try to push parts of the mechanisms/decisions, from the hardware to the software stack. One example of this would be to try to utilize the compiler to provide information about the code that is not easy for the hardware predictors to uncover. Knowing the number of dependences that each thread has, or that a specific part of the code is a simple hammock, may help for example on deciding how to treat a TLS thread or when one should stop Multi-Path execution (as with Diverge-Merge Processors [42]). Moreover making the compiler aware of the underlying execution models may create opportunities for unsafe optimizations for the helper threads that will never commit.

In terms of the power consumed, most the profitability based power allocation scheme can be directly applied to explicitly parallel applications. Instead of allocating power based on whether a thread will commit or not, allocation can be done based on the estimated slack of a specific thread to a barrier, or based on the memory behavior. It would also be interesting to investigate whether for specific loops, it is better to spend the given power budget to have a small number of fast cores, or a larger number of slower ones. Deciding at runtime whether which approach to follow, may result to a much more energy efficient result.

Bibliography

[1] S. Abedinpour, B. Bakkaloglu, and S. Kiaei. "A Multi-Stage Interleaved Synchronous Buck Converter with Integrated Output Filter in a 0.18um SiGe Process." In *International Solid-State Circuits Conf.*, February 2006.

[2] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. "Multipath Execution: Opportunities and Limits" *Intl. Conf. on Supercomputing,* pages 101-108, July 1998.

[3] R. Allen and K. Kennedy. "Optimizing Compilers for Modern Architectures: A Dependence-Based Approach." *Morgan Kaufmann,* 2002.

[4] G. M. Amdahl. "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities." *Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press,* pages 483-485, 1967.

[5] J. L. Aragón, J. González, J. M. Garca and A. González. "Confidence Estimation for Branch Prediction Reversal." *Intl. Conf. on High Performance Computing,* pages 213-224, December 2001.

[6] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. M. Hwu. "Beating In-Order Stalls with 'Fea-Ficker' Two-Pass Pipelining." *Intl. Symp. on Microarchitecture,* pages 387-398, December 2003.

[7] A. Bhattacharjee and M. Martonosi. "Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors." *Intl. Symp. on Computer Architecture,* pages 290-301, June 2009.

[8] M. Black and M. Franklin. "Perceptron-Based Confidence Estimation for Value Prediction." *Intl. Conf. on Intelligent Sensing and Information,* pages 271-276, December 2004.

[9] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, P. Tu. "Parallel Programming with Polaris." In *Proc. of Computer*, pages 78-82, December 1996.

[10] S. E. Breach. "Design and Evaluation of a Multiscalar Processor." *PhD Thesis, Department of Computer Science and Engineering, University of Wisconsin-Madison,* 1998.

[11] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: a Framework for Architectural-Level Power Analysis and Optimizations." *Intl. Symp. on Computer Architecture*, pages 83-94, June 2000.

[12] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors." *IEEE Micro,*, vol. 20, no. 6, pages 26-44, November 2000.

[13] M. G. Burke and R. K. Cytron. "Interprocedural Dependence Analysis and Parallelization." *Intl. Conf. on Programming Language Design and Implementation,* pages 139-154, June 1986.

[14] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. "CAVA: Using Checkpoint-Assisted Value Prediction to Hide L2 Misses." *ACM Trans. on Architecture and Code Optimization,* vol. 3, no. 2, pages 182-208, June 2006.

[15] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. "Simultaneous Subordinate Microthreading (SSMT)." *Intl. Symp. on Computer Architecture,* pages 186-195, May 1999.

[16] R. S. Chappell, F. Tseng, Y. N. Patt, and A. Yoaz. "Difficult-Path Branch Prediction Using Subordinate Microthreads." *Intl. Symp. on Computer Architecture,* pages 307-317, 2002.

[17] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, M. Tremblay. “Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun’s ROCK Processor.” *Intl. Symp. on Computer Architecture,* pages 484-4955, June 2009.

[18] T. Chelcea and S. M. Nowick. “Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols.” *Conf. on Design Automation,* pages 21-26, June 2001.

[19] M. C. Chidester, A. D. George, and M. A. Radlinski. “Multiple-Path Execution for Chip Multiprocessors” *Journal of Systems Architecture,* pages 33-52, July 2003.

[20] B. Choi, L. Porter, and D. M. Tullsen. “Accurate Branch Prediction for Short Threads.” *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 125-134, March 2008.

[21] M. Cintra and J. Torrellas. “Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors.” *Intl. Symp. on High-Performance Computer Architecture*, pages 43-54, February 2002.

[22] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. “Speculative Precomputation: Long-Range Prefetching of Delinquent Loads.” *Intl. Symp. on Computer Architecture,* pages 14-25, June 2001.

[23] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry. “Tolerating Dependences Between Large Speculative Threads Via Sub-Threads.” *Intl. Symp. on Computer Architecture,* pages 216-226, June 2006.

[24] Intel Corp. “Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors.” http://download.intel.com/design/processor/applnots/320354.pdf.

[25] Intel Corporation. *Intel Core2 Duo Processors and Intel Core2 Extreme Processors for Platforms Based on Mobile Intel 965 Express Chipset Family Datasheet,* 2008.

[26] P. Damron, A. Fedorova, Y. Lev, V. Luchango, M. Moir and D. Nussbaum. "Hybrid Transactional Memory." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 336-346, October 2006.

[27] G. Dhiman,T. S. Rosing. "Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning." *Intl. Symp. on Low Power Electronics and Design*, pages 207-212, August 2007.

[28] J. Dundas and T. Mudge. "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss." *Intl. Conf. on Supercomputing,* pages 68-75, July 1997.

[29] S. Eyerman, L. Eeckhout, J. E. Smith. "Characterizing the Branch Misprediction Penalty." *Intl. Symp. on Performance Analysis of Systems and Software,* pages 48-28, March 2006.

[30] S. Gopal, T. N. Vijaykumar, J. E. Smith and G. Sohi. "Speculative Versioning Cache." *Intl. Symp. on High-Performance Computer Architecture,* pages 195-205, January 1998.

[31] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. "Confidence Estimation for Speculation Control." *Intl. Symp. on Computer Architecture,* pages 122-131, June 2001.

[32] L. Hammond, M. Wiley, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 58-69, October 1998.

[33] P. Hazucha, G. Schrom, H. Jaehong, B. Bloechel, P. Hack, G. Dermer, S. Narendra, D. Gardner, T. Karnik, V. De, and S. Borkar. "A 233-MHz 80%-87% Efficiency Four-Phase DC-DC Converter Utilizing Air- Core Inductors on Package." In *International Solid-State Circuits Conf.*, February 2005.

[34] T. Heil and J. E. Smith. "Selective Dual Path Execution." *Technical Report, Department of Electrical and Computer Engineering, University of Wisconsin-Madison,* November 1996.

[35] M. Herlihy and J. E. B. Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures." *Intl. Symp. on Computer Architecture*, pages 289-300, May 1993.

[36] M. D. Hill and M. R. Marty "Amdahl's Law in the Multicore Era." *Proc. of Computer*, pages 33-38, July 2008.

[37] D. R. Hower and M. D. Hill. "Exploiting Episodes for Lightweight Race Recording." *Intl. Symp. on Computer Architecture*, pages 265-276, June 2008.

[38] C. Iwama, N. D. Barli, S. Sakai, and H. Tanaka. "Improving Conditional Branch Prediction on Speculative Multithreading Architectures." *Euro-Par*, pages 413-417, August 2001.

[39] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. " Control Flow Speculation in Multiscalar Processors." *Intl. Symp. on High-Performance Computer Architecture*, pages 218-229, February 1997.

[40] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. "Recovery Requirements of Branch Prediction Storage Structures in the Presence of Mispredicted-Path Execution." *Intl. Journal of Parallel Programming*, vol. 25, 1997.

[41] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks. "System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators." *Intl. Symp. on High-Performance Computer Architecture*, pages 123-134, February 2008.

[42] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. "Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths." *Intl. Symp. on Microarchitecture*, pages 53-64, December 2006.

[43] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martínez. "Checkpointed Early Load Retirement." *Intl. Symp. on High-Performance Computer Architecture*, pages 16-27, February 2005.

[44] A. Klauser, A. Paithankar, and D. Grunwald. "Selective Eager Execution on the Polypath Architecture" *Intl. Symp. on Computer Architecture,* pages 250-259, June 1998.

[45] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor." *Intl. Conf. on Supercomputing,* pages 85-92, June 1998.

[46] J. R. Larus and R. Rajwar. "Transactional Memory." *Morgan & Claypool Publishers,* 2006.

[47] A. W. Lim and M. S. Lam. "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms." *In Proc. of Parallel Computing,* p. 201-214, 1997.

[48] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. "Value Locality and Load Value Prediction." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 138-147, October 1996.

[49] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. "POSH: a TLS Compiler that Exploits Program Structure." *Symp. on Principles and Practice of Parallel Programming,* pages 158-167, March 2006.

[50] Gabriel H. Loh. "Simulation Differences Between Academia and Industry: A Branch Prediction Case Study." *Intl. Symp. on Performance Analysis of Software and Systems,* pages 21-31, March 2005.

[51] P. Macken, M. Degrauwe, M. V. Paemel, and H. Oguey. "A Voltage Reduction Technique for Digital Systems." *Intl. Solid-State Circuits Conf.,* pages 238239, February 1990.

[52] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. "A Voltage Reduction Technique for Digital Systems." *IEEE Intl Solid-State Circuits Conference,* pages 238-239, February 1990.

[53] P. Marcuello and A. González. "Clustered Speculative Multithreaded Processors." *Intl. Conf. on Supercomputing*, pages 365-372, June 1999.

[54] P. Marcuello and A. González. "A Quantitative Assessment of Thread-level Speculation Techniques." *Intl. Parallel and Distributed Processing Symp.*, pages 595-602, May 2000.

[55] S. McFarling. "Combining Branch Predictors." *WRL Technical Note TN-36*.

[56] P. Montesinos, L. Ceze, and J. Torrellas. "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently." *Intl. Symp. on Computer Architecture*, pages 289-300, June 2008.

[57] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. "Dynamic Speculation and Synchronization of Data Dependence." *Intl. Symp. on Computer Architecture*, pages 181-193, May 1997.

[58] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. "Runahead Execution: An Alternative to Very Large Instruction Windows." *Intl. Symp. on High-Performance Computer Architecture*, pages 129-140, February 2003.

[59] Y. Panov and M. Jovanovic. "Design Considerations for 12-V/1.5-V, 50-A Voltage Regulator Modules." *Transactions on Power Electronics*, 16(6), November 2001.

[60] R. L. Plackett and J. P. Burman. "The Design of Optimum Multifactorial Experiments." *In Proc. of Biometrica*, vol. 33, no. 4, pages 305-325, June 1946.

[61] C. G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. González, and D. M. Tullsen. "Mitosis Compiler: an Infrastructure for Speculative Threading Based on Pre-Computation Slices. " *Intl. Conf. on Programming Language Design and Implementation*, pages 269-279, June 2005.

[62] M. Ramsay, C. Feucht, and M. H. Lipasti. "Exploring Efficient SMT Branch Predictor Design." *Workshop on Complexity-Effective Design*, June 2003.

[63] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. "Tasking with Out-Of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation." *Intl. Conf. on Supercomputing,* pages 179-188, June 2005.

[64] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. "SESC simulator." *http://sesc.sourceforge.net*.

[65] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. "Design Trade-offs for the EV8 Branch Predictor." *Intl. Symp. on Computer Architecture,* pages 295-306, June 2002.

[66] A. Seznec. "Analysis of the OGEHL predictor." *Intl. Symp. on Computer Architecture,* pages 394-405, June 2005.

[67] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. "Temperature-Aware Microarchitecture." *Intl. Symp. on Computer Architecture,* pages 2-13, June 2003.

[68] G. S. Sohi, S. E. Breach and T. N. Vijaykumar. "Multiscalar Processors." *Intl. Symp. on Computer Architecture,* pages 414-425, June 1995.

[69] J. G. Steffan, C. B. Colohan and T. C. Mowry. "Architectural Support for Thread-Level Data Speculation." *Tech. Rep.* CMU-CS-97-188, November 1997.

[70] J. G. Steffan, C. B. Colohan, A. Zhai and T. C. Mowry. "A Scalable Approach to Thread-Level Speculation." *Intl. Symp. on Computer Architecture,* pages 1-12, June 2000.

[71] J. G. Steffan and T. C. Mowry. "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization." *Intl. Symp. on High-Performance Computer Architecture,* pages 2-13, February 1998.

[72] "SPEC Benchmark Suite." *http://www.spec.org/cpu2000*.

[73] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. "Slipstream Processors: Improving Both Performance and Fault Tolerance." *Intl. Conf. on Architectural*

Support for Programming Languages and Operating Systems, pages 257-268, October 2000.

[74] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical report, Compaq Western Research Lab., 2006.

[75] G. Tournavitis, Z. Wang, B. Franke and M. F. P. O'Boyle. "Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based Mapping." *Intl. Conf. on Programming Language Design and Implementation*, pages 177-187, June 2009.

[76] J. Tuck, W. Liu, and J. Torrellas. "CAP: Criticality Analysis for Power-Efficient Speculative Multithreading." *Intl. Conf. on Computer Design*, pages 409-416, October 2007.

[77] J. Tuck, L. Ceze, and J. Torrellas. "Scalable Cache Miss Handling for High Memory-Level Parallelism." *Intl. Symp. on Microarchitecture,* pages 409-422, December 2006.

[78] DṀ. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor." *Intl. Symp. on Computer Architecture,* pages 191-202, May 1996.

[79] F. Warg. "Techniques to Reduce Thread-Level Speculation Overhead." *PhD Thesis, Department of Computer Science and Engineering, Chalmers University,* 2006.

[80] J. Wibben and R. Harjani. "A High Efficiency DC-DC Converter Using 2nH On-Chip Inductors." *Symp. on VLSI Circuits*, 2007.

[81] W. Wu, N. Lee, and G. Schuellein. "Multi-Phase Buck Converter Design with Two-Phase Coupled Inductors." In *Applied Power Electronics Conference and Exposition*, 2006.

[82] P. Xekalakis, N. Ioannou and M. Cintra. "Combining Thread Level Speculation, Helper Threads, and Runahead Execution." *Intl. Conf. on Supercomputing,* pages 410-420, June 2009.

[83] J. J. Yi, D. J. Lilja and D. M. Hawkins. "A Statisticaly Rigorous Approach for Improving Simulation Methodology." *Intl. Symp. on High Performance Computer Architecture*, pages 281-290, February 2003.

[84] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry "Compiler Optimization of Scalar Value Communication Between Speculative Threads." *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems,* pages 171-183, October 2002.

[85] C. Zilles and G. Sohi. "Execution-Based Prediction Using Speculative Slices." *Intl. Symp. on Computer Architecture,* pages 2-13, June 2001.