THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Techniques to Reduce Thread-Level Speculation Overhead

Fredrik Warg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2006

Techniques to Reduce Thread-Level Speculation Overhead
Fredrik Warg
ISBN 91-7291-803-9

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2485
ISSN 0346-718X

Technical report 18D
Department of Computer Science and Engineering
Research group: High-Performance Computer Architecture

Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

URI: `http://www.cse.chalmers.se/`
Author email address: `fredrik@warg.org`

Printed by Chalmers Reproservice
Göteborg, Sweden 2006

Techniques to Reduce Thread-Level Speculation Overhead

Fredrik Warg
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The traditional single-core processors are being replaced by chip multiprocessors (CMPs) where several processor cores are integrated on a single chip. While this is beneficial for multithreaded applications and multiprogrammed workloads, CMPs do not provide performance improvements for single-threaded applications. Thread-level speculation (TLS) has been proposed as a way to improve single-thread performance on such systems. TLS is a technique where programs are aggressively parallelized at run-time – threads speculate on data and control dependences but have to be squashed and start over in case of a dependence violation. Unfortunately, various sources of overhead create a major performance problem for TLS.

This thesis quantifies the impact of overheads on the performance of TLS systems, and suggests remedies in the form of a number of overhead-reduction techniques. These techniques target run-time parallelization that do not require recompilation of sequential binaries. The main source of parallelism investigated in this work is module continuations, i.e. functions or methods are run in parallel with the code following the call instruction. Loops is another source.

Run-length prediction, a technique aimed at reducing the amount of short threads, is introduced. An accurate predictor that avoids short threads, or dynamically unrolls loops to increase thread lengths, is shown to improve speedup for most of the benchmarks applications. Another novel technique is *misspeculation prediction*, which can remove most of the TLS overhead by reducing the number of misspeculations.

The interaction between thread-level parallelism and instruction-level parallelism is studied – in many cases, both sources can be exploited for additional performance gains, but in some cases there is a trade-off. Communication overhead and memory-level parallelism are found to play an important role. For some applications, prefetching from threads that are squashed contributes more to speedup than parallel execution. Finally, faster inter-thread communication is found to give simulataneous multithreaded (SMT) processors an advantage as the basis for TLS machines.

Keywords: Computer architecture, thread-level speculation, chip multiprocessors, multithreaded processors, speculation overhead, performance evaluation.

Publications

Parts of this thesis are based on the following publications:

- Fredrik Warg and Per Stenstrom, Limits on Speculative Module-level Parallelism in Imperative and Object-oriented Programs on CMP Platforms, in *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*, pages 221-230, September 2001.
- Fredrik Warg and Per Stenstrom, Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction, in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 12 (abstract, full paper on accompanying cd), April 2003.
- Fredrik Warg and Per Stenstrom, Reducing Misspeculation Overhead for Module-Level Speculative Execution, in *Proceedings of the 2005 ACM International Conference on Computing Frontiers (CF 2005)*, pages 289 - 298, May 2005.

Publication not part of the thesis:

- Magnus Ekman, Fredrik Warg, and Jim Nilsson, An In-Depth Look at Computer Performance Growth, TR04-9, Department of Computer Engineering, Chalmers University of Technology, October 2004. Also appears in *ACM SIGARCH Computer Architecture News*, Volume 33, Issue 1 (March 2005), pages 144 - 147, 2005.

Contents

Preface

"I have not failed. I've just found 10 000 ways that won't work."
– Thomas Alva Edison (1847-1931)

In my experience, the words of Thomas Edison quoted above nicely captures a very useful way to think when doing research.[1] Many "bright" ideas won't work as well as you hoped they would, but at least you'll know why – knowledge that may eventually lead you to the right solution. Edison was a controversial man, but also a great inventor and someone I admired as a kid. There probably aren't that many kids who are fans of long dead engineers, but then again, most kids don't end up pursuing a Ph.D in computer engineering as adults. At least in my case, I'm sure these two things are related.

I want to take this opportunity to thank a number of people for their help, support, and friendship during my time at Chalmers:

- First of all, my advisor Per Stenström, for sharing his broad knowledge of research and computer architecture, and for remaining encouraging and enthusiastic about my project even when I felt I wasn't getting anywhere.
- Fredrik Dahlgren, my master's thesis advisor, and the person who brought me to Chalmers in the first place.
- Martin Thuresson and Tom Ashby for reading and providing valuable comments on parts of this thesis.
- Past and present members of the high-performance computer architecture group (in no particular order): Peter, Jonas(x2), Magnus(x2), Jim, Jochen, Charlotta, Martin(x2), Thomas, Waliullah, and Mafijul.
- My fellow Ph.D. students and other employees at computer engineering, who have all contributed to an enjoyable work environment.

[1] I said a useful way, not necessarily always an easy way...

Last but not least, a big thanks to my family and friends for all your support over the years – and most important of all, to Helena, for your love and support.

This work has been funded by NUTEK (Swedish Industrial Board for Technical Development), SSF (Swedish Foundation for Strategic Development), VR (Swedish Research Council), and the SARC project (funded by the European Commission under the FET program). Equipment grants from Sun Microsystems Inc. and access to the Swegrid computational grid operated by SNIC (Swedish National Infrastructure for Computing) have been indispensable when running the many simulations needed to obtain the results presented in this thesis.

1 Introduction

Traditionally, the vast majority of all computers have had a single processor core managing all the computations. More powerful multiprocessor computers have been built by connecting several single-core processor chips. These multiprocessor machines have mostly been used as server machines. Typical multiprocessor workloads are serving many clients in parallel, or computations for a limited set of specialized tasks, where it has been possible to customize the applications to take advantage of many processor cores. For general-purpose computing, multiprocessors have not been very useful. Many applications are written to serve a single user or perform some computation which is not easy to split among several processors. These *single-threaded applications* are written to run on a single processor core.

Performance improvements for single-threaded applications have so far been accomplished with increasingly advanced processor cores. The improvements achieved with a single core during the past decades can hardly be described as anything other than spectacular. The SPEC CPU integer benchmark application suites (SPEC CINT) have long been a broadly accepted way to measure general-purpose processor performance. Figure 1.1 shows how single-thread performance has developed during the past decades.[1] The average annual performance growth for the SPEC CINT bench-

[1] This is an updated version of a graph from [EWN04]. It shows performance growth based on official results from three versions of SPEC CINT. The results have been normalized to the same relative performance scale.

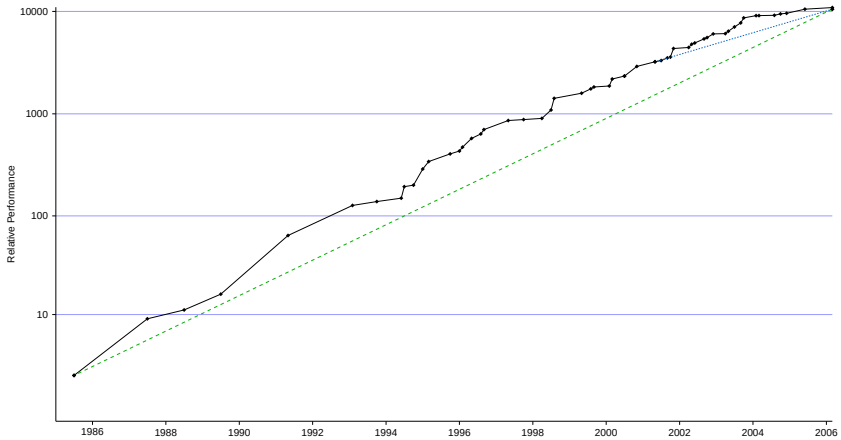


Figure 1.1: *Single-thread performance growth 1985-2006.*

marks is about 52% for the twenty years between mid 1985 and today (coarse dotted line). The rapid growth rate has been made possible by advances in VLSI technology, compilers, and computer architecture.

The bad news is that the major sources of this performance growth, namely increased clock speed and better ways to exploit instruction-level parallelism (ILP), are no longer improving at the pace they used to. Figure 1.1 also shows that the growth rate has gradually been slowing down for many years. For instance, if the performance growth curve is split five years ago, the annual growth in the first part, 1985-2001, is an impressive 60%. During the last five years, 2001-2006, the average annual growth rate has been a more modest 29% (fine dotted line). Although we might still see improvements from single-core processors, several factors suggest that the trend of declining single-thread performance growth for general-purpose processors is going to continue in the near future.

A major hurdle is the approaching physical limits for the CMOS technology used when manufacturing microprocessors. Modern processors with high transistor density and high clock frequencies generate much heat. High-performance single-thread microprocessors are running into a thermal wall, which inhibits further increases in operating frequency.

On the architecture side, recent efforts to extract more instruction-level parallelism, i.e. running neighboring independent instructions in parallel within the processor core, have not been as successful as before. Designers are facing diminishing returns when trying to expand the instruction window in order to find more independent instructions to run in parallel. Larger instruction windows have tradition-

ally come at the expense of more complex control logic, and complexity makes it more difficult to attain high clock frequencies. Part of the problem with exploiting ILP is also attributable to high memory latencies and imperfect branch prediction, which limits the useful instruction window size, problems that are only exacerbated in deeply pipelined high clock frequency designs. Thus, designers face a trade-off between ILP and clock frequency. Complex designs are typically also less efficient. It is well known that the size and complexity of processor designs have been increasing at a faster pace than the performance.

To summarize, increasing clock frequency is difficult due to heat problems, and exploiting more ILP is both architecturally difficult and comes at the price of increasingly complex and inefficient designs. Another important consideration is energy efficiency, and neither high frequency nor overly complex processors excel in that respect.

1.1 Multithreaded Systems

While there are likely still gains to be found in frequency scaling and ILP, it is clear that we need alternative ways to build better microprocessors. To this end, microprocessor manufacturers are now aggressively pursuing on-chip thread-level parallelism (TLP), i.e. running several independent threads of control on the same chip. There are two major classes of such chips: the chip multiprocessor (CMP) [ONH+96], where multiple independent processor cores are integrated on the same chip; and simultaneous multithreading (SMT) [TEL95], where multiple threads of control share some of the resources in a single processor core. It is also possible to combine these two techniques to build chip multiprocessors where each core is an SMT processor, an approach that has been adopted by the Sun Niagara [KAO05] and Rock [CCYT05] chips and IBM POWER 5 [KST04].

I will collectively refer to these designs as *multithreaded processors*. Figure 1.2 shows an example organization for a multithreaded chip. The advantage of this approach is the potential to achieve higher aggregate performance by running multiple threads of control in parallel, instead of relying on increased per-thread ILP and clock frequency scaling. Multithreaded processors constitute the architectural framework of this thesis.

For desktop computers, adding a second processor core will typically improve application performance, since modern operating systems run a variety of tasks at any given moment, i.e. a multiprogrammed workload. Background tasks can be offloaded to the second core, giving the primary task more processor time. This gain will not scale when transitioning to even more threads though, as desktop computers rarely run a large number of computationally intensive tasks.

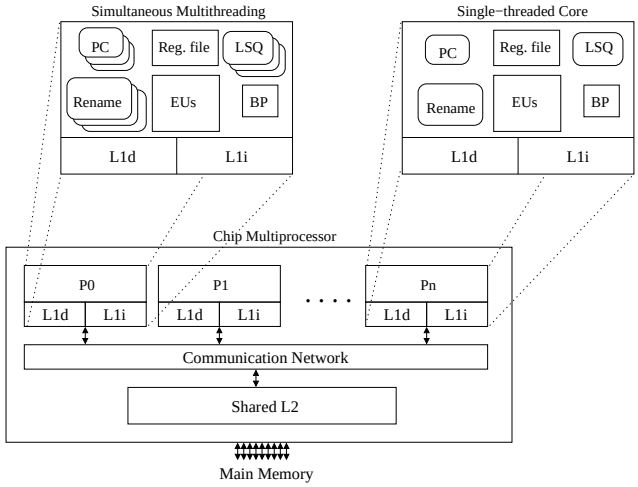


Figure 1.2: *An example multithreaded processor. The chip contains several processors and a shared level two cache tied together with an on-chip interconnect. The chip could contain either SMT or traditional processor cores.*

Multithreaded chips are a good solution for cost effective and energy efficient server processing. Many typical server applications contain plenty of parallelism that will naturally scale with the problem size. A good number of scientific applications exhibit similar properties. These are sometimes called *embarrassingly parallel* problems. Such problems can typically be successfully parallelized by a programmer using one of many available parallel programming abstractions.

However, far from all applications are embarrassingly parallel; on the contrary, some problems are very difficult to parallelize. This class does not have a similar witty name yet, but perhaps *annoyingly sequential* would fit the bill. Unfortunately, multithreaded chips will not improve performance for this class of applications.

Another drawback is that parallel programming places a larger burden on the programmer; designing a parallel program is more error prone and debugging is significantly more difficult than constructing a traditional sequential application. Writing efficient parallel programs is harder still. Manually making sure the programs are free of races, live-locks and deadlocks, have good load balancing, and low communication overhead runs contrary to the rapid software development methodologies many have come to expect. In short, it is not a realistic expectation that the majority of program development in the future will be parallel programming given the requirements of today's parallel programming methodologies.

Finally, we have a vast library of legacy software that would be too costly to re-

implement in order to reap the benefit of multithreaded processors. All this points to a need for easier methods to benefit from thread-level parallelism.

Automatic parallelization with compilers is one alternative to explicit parallel programming. Such compilers analyze the source code of an application and try to uncover parallelism that can be exploited. An application can be parallelized if the compiler finds sections which it can prove to have no dependences between them. While this is useful, a fundamental problem is that compilers lack information about the input data at compile time. This means the compiler can not always prove that potentially parallel threads are in fact independent. If it can not prove independence, the code can not be parellelized. For regular numeric applications, parallelizing compilers have been somewhat successful; however, for applications with more complex control flow and data access patterns, they have not.

1.2 Thread-Level Speculation

The goal of thread-level speculative execution techniques is to speed up single-threaded applications on shared memory systems with resources to execute multiple threads in parallel. Thus, thread-level speculation (TLS) can be combined with chip multiprocessors, simultaneous multithreaded processors, or traditional shared memory multiprocessors. Single-threaded applications are not originally written to take advantage of multiple processors/threads. TLS overcomes this limitation by automatically splicing up the sequential application into threads, running several of these threads in parallel. This thesis investigates thread-level speculation on multithreaded architectures, with a focus on chip multiprocessors.

Figure 1.3 (left) shows an example where a piece of code, executing instructions *I1* through *In*, is split into three separate threads, *T1*, *T2* and *T3*, executing in parallel. In order to avoid the problem facing parallel compilers, TLS drops the requirement that threads are provably independent. Instead, threads are optimistically spawned where one has reasons to believe they will be independent, or have dependences that can be resolved at run time. That is, the threads are spawned speculatively, assuming a risk that the threads might prove not to be independent.

In the example on the right hand side in Figure 1.3, thread *T3* reads memory location *a* which is computed and written back to memory in thread *T2*. When running the threads in parallel, that value is not available when *T3* needs it. The thread will read an out-of-date value, and the results computed with this value are erroneous. When such a problem occurs, called a data dependence violation, the system must be able to recover from the erroneous results it has caused. Eventually, the application must end up with the same result the original sequential execution would yield. TLS systems typically achieve this by restarting threads which suffer from a dependence violation, as shown in the figure.

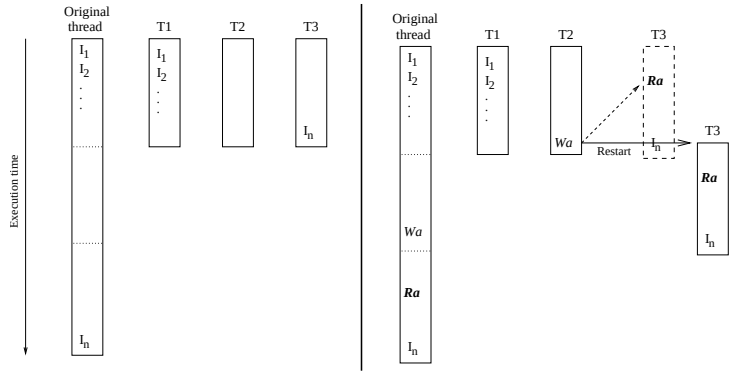


Figure 1.3: *Example of thread-level speculation (left) and a dependence violation (right).*

TLS requires support mechanisms, either in software or hardware, for starting and finalizing (or committing) threads, detecting dependence violations and recovering from violations. In addition, some method for determining when and where to spawn off new threads is necessary. A common source of parallelism is to spawn new threads for successive loop iterations. While loop parallelism is used in parts of this thesis, the focus is on module-level parallelism. Module-, or subroutine parallelism is exploited by starting a new thread at the module continuation, i.e. the code after a subroutine call, as the original thread executes the called subroutine.[2] This means the subroutine and the code following the subroutine will be executed in parallel.

This work builds upon the large body of work done on TLS implementations that is presented in Chapter 2. Early proposals for TLS systems as an extension of a CMP, which provided inspiration for the work in this thesis, are the Hydra project [HWO98] and STAMPede [SM98]. It is common among the TLS systems to rely on the compiler to adapt the code to the TLS architecture. This thesis, on the contrary, investigates techniques which can be applied to sequential binaries. That is, the techniques do not require access to the source code of the application being parallelized.

[2]Many different names have been used for this kind of parallelism. Examples are procedure-, method-, function-, and subroutine-level parallelism. I have used the term module-level parallelism in previously published papers instead of the other terms which are associated with various programming language constructs. The motivation was to stress that the technique is not language specific. I will continue to use this name throughout this thesis in order to be consistent with the earlier work. My definition of module-level parallelism is that threads are spawned at the instruction following a call in the instruction stream, also called the module continuation, and ends at return instructions.

1.3 Problem Statement

While a program can be parallelized without concern for correctness with the TLS model, it is still imperative to do a good job with parallelization in order to achieve high performance. When dependence violations occur, one or several threads need to roll back execution to a known correct state and re-execute with the correct input values. On a perfect machine, with no overheads, TLS parallelized applications would never run slower than their sequential counterparts, and would run faster whenever there are threads which can execute in parallel. Unfortunately we have to deal with a number of real-world limitations to this ideal model. Rolling back execution, managing threads, and the communication between threads will create overhead. This means the efficiency of TLS parallelization comes down to an efficient implementation of TLS support as well as sensible policies for selecting and spawning speculative threads.

The central question addressed in this work is:

> *What is the nature and quantity of overheads incurred by thread-level speculation on realistic multithreaded processor models, and how can these overheads be reduced?*

The advantages of reducing overhead are two-fold, improving the speedup gained over sequential execution and avoiding excessive amounts of wasted execution due to dependence violations. Wasted execution is unwelcome both from an energy efficiency perspective and because the resources that were tied up for no benefit could have been better used by other threads running in the system.

Thread-level speculation techniques can be divided into two major groups: dynamic techniques aiming to parallelize unmodified single-threaded applications at run-time, and static techniques requiring compiler support to improve the parallelism. This thesis is squarely focused on dynamic techniques.

1.4 Methodological Approach

In order to answer the research questions, I use a simulation framework which models a chip multiprocessor with TLS support. In order to gain an understanding of the nature of speculation overheads, the work is carried out with a number of increasingly more detailed machine models. At first, the models use an idealized speculation system and disregard the impact of communication overheads and complex out-of-order processor cores, so as to understand the inherent parallelism in applications. In the second part of the thesis, the focus is shifted to architectural implementation issues and detailed processor and memory hierarchy models are used.

This approach allows me to study the various forms of overhead and other performance-limiting bottlenecks one at a time. The first models are limited only by the inherent parallelism in the benchmark applications. With the more detailed models, an increasing number of architectural constraints begin to affect the performance.

1.5 Contributions

The main contributions of my work are:

1. Quantification of the inherent module-level parallelism in a number of numeric applications written in both imperative (C) and object-oriented (Java) programming styles. Furthermore, the performance impact of a number of design constraints are identified and quantified. The main findings are that module-level parallelism typically does not scale above 4-8 threads and therefore fits a small-scale CMP. In addition, dependence violations are prevalent and the granularity of modules does not match the overhead in a typical CMP. Much of this work has been published in [WS01]. [Chapter 3]

2. I introduce the concept of *run-length prediction*, a technique aimed at reducing the number of small module-level threads spawned using indiscriminate speculation, i.e. when threads are spawned for all modules. Run-length prediction is a history based technique which predicts the final size of a thread before it has been spawned, and prevents a thread from being spawned if the predicted size is below a certain threshold. This technique is shown to improve the performance of module-level speculation considerably. This work has been published in [WS03]. [Chapter 4]

3. I investigate *parallel overlap prediction* as a means to reduce the total overhead and remove threads that have a small execution overlap with its parent, i.e. threads that do not contribute much to the speedup but increase the amount of wasted execution. The results show that the technique indeed improves the situation, but the remaining overhead in wasted execution is still significant. [Chapter 5]

4. A second technique aimed at reducing wasted execution is introduced. *Misspeculation prediction* is an attempt to decrease the overhead by avoiding misspeculations. Consequently, the technique aims at predicting if a potential new thread will fall prey to a misspeculation. Module calls that are likely to misspeculate are marked as non-parallel and prevented from spawning new threads. The principal findings are that misspeculation prediction can remove

most of the execution overhead with only a small negative performance impact due to removing useful parallelism. In addition, it is found that misspeculation prediction can be detrimental to performance for applications which do not have an excess of execution overhead to begin with. In this situation, a complementary technique for selectively enabling and disabling misspeculation prediction is found to be effective. This work has been published in [WS05]. [Chapter 6]

5. A systematic analysis of the impact of architectural implementation issues is provided. Specifically, the effects of communication overhead and memory latencies, pipelining, issue width, and branch prediction are studied. The analysis shows that issue-width, perhaps somewhat surprisingly, does not affect the speedup provided by TLS for most applications; ILP is often orthogonal to TLS, or not significant enough to affect TLS parallelism negatively. There is a trade-off between ILP and TLS only for some high-ILP applications. Another insight is that memory access latency typically do not have a negative impact on speedup compared to the inherent parallelism. On the contrary, the prefetching effect from squashed threads and memory-level parallelism from successful threads both contribute to increase the benefit of thread-level speculation when taking communication overhead into account. The prefetching effect occurs when the memory accesses issued by threads that are later squashed work as useful prefetches for threads executing later and access the same locations. [Chapter 8]

6. Evaluation of the potential of *deferred squash* for data speculative threads. Deferred squash means that a thread is not immediately restarted after a violation, but allowed to continue executing until it ends or becomes non-speculative. This is done to improve the prefetching effect. Initial measurements, although with a somewhat optimistic model, show that deferred squash is a promising technique for the applications that already gain speedup due to the prefetching effect. [Section 8.4.3]

7. Run-length prediction is extended to loops, and modified to double as a dynamic loop unrolling mechanism. This new technique is shown to improve the performance of loop-level speculation for several applications. Furthermore, I confirm that run-length prediction for modules works when communication overhead is taken into account, as well as with pipelined and wide-issue processors. For SMT processors, the key observation is that, since thread management overheads are lower for an SMT than a CMP, the effect of run-length prediction is smaller, and the technique should be used with more care to avoid loss of parallelism. [Section 7.3.4 and Section 8.5]

8. The impact of communication overhead on misspeculation prediction is evaluated. The conclusions are that misspeculation prediction continues to successfully remove overhead, and improves performance for some applications. However, a drawback is that the technique does not consider the effect of prefetching and memory-level parallelism. Due to this omission, the TLS speedup is decreased or eradicated for applications relying on these effects. Therefore, misspeculation prediction needs to be modified to work well in a realistic system. [Section 7.3.4 and Section 8.6]

9. A comparison of TLS performance on chip multiprocessors and SMT processors. The main conclusion is that, given an equal total issue width, the SMT processor typically performs better due to lower thread-management overhead and lower overhead for communication between threads. However, the shared L1 cache may be a bottleneck for the SMT processor unless size and associativity can be scaled with the number of threads. [Section 9.2]

10. TLS for a system with only one speculative thread is evaluated. I show that a significant amount of the available parallelism can be exploited in such a machine. Even if only speculative state for one thread at a time can be handled, which means preemption of idle threads is not possible, the performance impact is small with only one speculative thread. This implementation would greatly simplify the TLS hardware, however, the potential for further performance improvements is small. [Section 9.3]

1.6 Thesis Organization

Thread-level speculation is an established research area and my work builds upon a significant body of previous work. Chapter 2 serves as an introduction to thread-level speculation in general, introduces important work in the area, and discusses major challenges with implementing TLS. Furthermore, much of the terminology used in the remaining chapters is introduced. Readers up-to-date with the TLS literature and terminology may skip this chapter.

The rest of the thesis contains the central contributions of my work. In Chapters 3 to 6 TLS overhead is studied with a focus on the inherent parallelism in the applications and the overhead from the speculation system itself. Then, in Chapters 7 to 9 the focus is shifted to implementation issues such as processor model, memory hierarchy, and speculation system. Chapter 3 investigates the potential of module-level parallelism and identifies the key bottlenecks. The subsequent three chapters each present a novel technique that can reduce the overhead introduced by thread-level speculative execution: run-length prediction in Chapter 4, parallel overlap prediction in Chapter 5 and finally misspeculation prediction in Chapter 6.

In Chapter 7, a detailed simulation framework supporting out-of-order execution, a multi-level memory system, and simultaneous multithreading is introduced. The major parts of an implementable speculation system are presented. This simulation model is used to gain insights into the impact of wide-issue out-of-order processors, communication overhead, and limitations of an implementable speculation system. These issues are investigated in Chapter 8. This chapter also contains a validation and expansion of the investigation of run-length and misspeculation prediction, and introduces deferred squash. Chapter 9 investigates alternative machine organizations. First, the CMP model used throughout the thesis is compared to TLS on an SMT processor. Then, the performance potential of a simplified system with only one speculative thread is explored.

Finally, Chapter 10 contains concluding remarks and thoughts about the future of research on thread-level speculation.

2

TLS: Models and Implementations

In order to implement a thread-level speculation system, there are a number of key implementation issues to solve. This chapter presents the thread-level speculative execution model in detail, and discusses its implementation.

The chapter serves several purposes. It introduces thread-level speculation for readers who are not familiar with this technique, and establishes the TLS terminology which will be used throughout the remaining chapters. Another purpose is to briefly survey research on the implementation of thread-level speculation systems. Finally, the feasibility of constructing a computer system with thread-level speculation support is established.

Section 2.1 introduces the chip multiprocessor, which is the base architecture for the machine models used in the following chapters. Section 2.2 presents the TLS execution model and lists the design issues that need to be addressed in a working TLS system; it also discusses how some TLS projects have addressed these issues. Section 2.3 summarizes additional TLS projects with respect to the capabilities of and design choices made for each architecture, and Section 2.4 discusses research on the related topic transactional memory. While this thesis considers only dynamic techniques, Section 2.5 will look at compiler techniques for improving TLS performance.

The survey of TLS architectures in Section 2.3 is not vital for the understanding of the remaining chapters and may be considered optional reading. Readers familiar with the TLS literature and terminology may skip this chapter altogether.

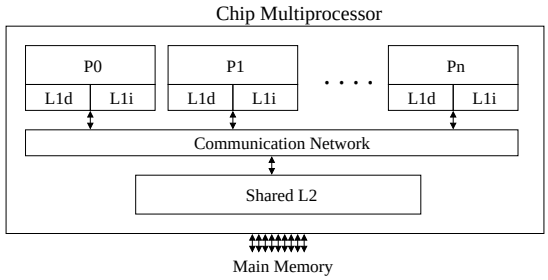


Figure 2.1: *Chip multiprocessor with* n *cores, and a shared level two cache.*

2.1 Chip Multiprocessors

Chip multiprocessors are multiprocessor computers on a single chip. The baseline architecture envisioned in this thesis is based on a CMP with an architecture like the one shown in Figure 2.1. A number of processor cores, P1 through P*n* in the figure, each with their own separate level one data (dL1) and instruction (iL1) caches are connected with an on-chip interconnect. In addition, the chip contains a larger level two (L2) cache which is shared among the cores.[1] Finally, a memory interface for access to off-chip main memory is connected to the L2 cache.

Chip multiprocessors are commercially available from several vendors. They share the common characteristic of multiple cores on a single chip, but there are some variations in their architecture. For instance, the IBM POWER 4 and POWER 5 [KST04] chips incorporate two relatively advanced superscalar cores, and the POWER 5 core also supports simultaneous multithreading. These designs use a crossbar switch to connect the cores with a shared level 2 cache. The AMD Opteron and Athlon64 X2 chips are dual core CMPs as well, but without a shared on-chip cache. Instead, the interconnect is on the level below L2 in the memory hierarchy, and only off-chip I/O is shared. The Sun Ultrasparc T1 (Niagara) chip [KAO05] integrates up to 8 simple processor cores and a shared L2 on a chip. The cores connect to the 4-way banked L2 with a crossbar switch.

The goal of multiprocessor machines is to increase throughput by running independent threads in parallel, while retaining the possibility of efficiently communicating between these threads when needed. This enables some time-consuming tasks to be split into multiple units that are largely independent but coordinate the work by communicating data. In shared-memory multiprocessors, which TLS builds upon, communication is performed through main memory. That is, threads running on different processors can access the same memory locations. Typically, some sort

[1] The terms processor and core are used interchangeably throughout the thesis.

of programming primitives are used to synchronize the communication, for instance barriers and locks. In between synchronization points the threads can run independently.

Compared to traditional multiprocessors, the inter-core communication in CMPs is much faster, since it can be handled on-chip through the shared bus and the shared L2 cache. This potentially enables them to exploit finer grained parallelism, which makes them a good match for TLS techniques.

Cache Coherence

For performance reasons, in most shared-memory multiprocessor systems each processor has its own private caches. The caches are used to store local copies of a subset of the main memory. Since the same location may be used by more than one processor, it is necessary to make sure the caches are coherent, i.e. that all copies of a memory location contain the same value. This is typically solved with a cache coherence protocol. As many TLS projects use extensions to a cache coherence protocol to implement key TLS functionality, it is important to be aware of how a typical coherence protocol works.

Figure 2.2 shows a simple snoopy coherence protocol used for a shared bus interconnect. A snoopy protocol has a cache controller that listens both to the bus and the processor the cache is attached to. The controller updates the cache state based on accesses from its own processor as well as requests from other processors, if the memory location being requested is stored in its cache.

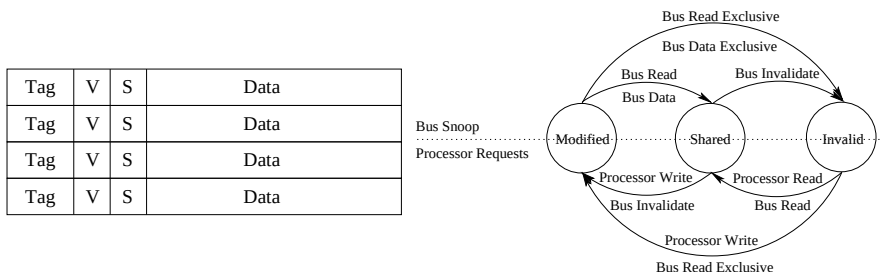


Figure 2.2: *The MSI cache coherence protocol.*

The cache is organized as an array of cache lines, or cache blocks, illustrated on the left hand side of the figure. Each line contains data from a number of consecutive addresses, a tag identifying the memory addresses it contains, and some control bits encoding the state of the data in the cache line. The state diagram to the right in the figure shows that a cache line can be in one of three distinct states: modified, shared,

or invalid. This is called an MSI protocol. The cache controller modifies the state of the cache line as a result of a number of events:

- If the processor reads and there is a tag match (i.e. the data is in the cache), the request is a cache hit and the data can immediately be returned to the processor.
- If the processor reads and there is no tag match, the request misses and is sent out on the bus. When the data is received, the line is set to the shared state.
- If the processor stores, there is a tag match and the cache line is in the modified state, the data can be stored directly to that line.
- If the processor stores, there is a tag match and the cache line is in a shared state, the cache issues an invalidate bus request and changes the state to modified. All other caches invalidate the cache line if they have a copy.
- If the processor stores and there is no tag match, the cache issues a read exclusive bus request. When the data is received, the cache line is set to modified. Other caches will invalidate their copies.

In short, there may be many copies of a cache line as long as they are only read, but as soon as a processor writes to a location, all other copies are invalidated. There are more elaborate implementations of this basic scheme, but it is beyond the scope of the thesis to cover them here. A comparison of coherence protocols is presented in a survey by Stenstrom [Ste90]. A more detailed description of cache coherence can be found in *Parallel Computer Architecture: A Hardware/Software Approach* [CSG99].

The described protocol belongs to the class of write-back invalidation based protocols. Write-back protocols store modified cache blocks locally, and do not propagate the updated data to lower levels of the memory hierarchy until they are evicted from the cache to make space for new data. The other alternative is a write-through cache where all updates are immediately written to lower levels of the memory system. An invalidation based protocol will send out invalidation requests to the other caches when a cache line in the shared state is modified, while an update based protocol will send out the updated data to other caches when a store to a shared cache line occurs.

The cache lines to the left in the figure show how the three states are encoded: each line has a valid (*v*) bit and a store (*s*) bit.[2] If both bits are cleared, the cache line is invalid. With only the valid bit set, the line is in the shared state, and if both valid and store are set, the state is modified.

[2] The store bit is sometimes called the dirty bit or modified bit.

2.2 Thread-Level Speculation

In the thread-level speculative execution model, fine-grained *speculative threads* are spawned from a single sequential program. The speculative threads run in parallel with the original, or *non-speculative thread*.

An advanced conventional processor can manage up to a few hundred instructions in-flight at any point in time, the so-called instruction window. From the instructions that are decoded but not yet executed, the processor can choose independent instructions to execute in parallel. In order to increase the potential parallelism, the instruction window needs to be larger.

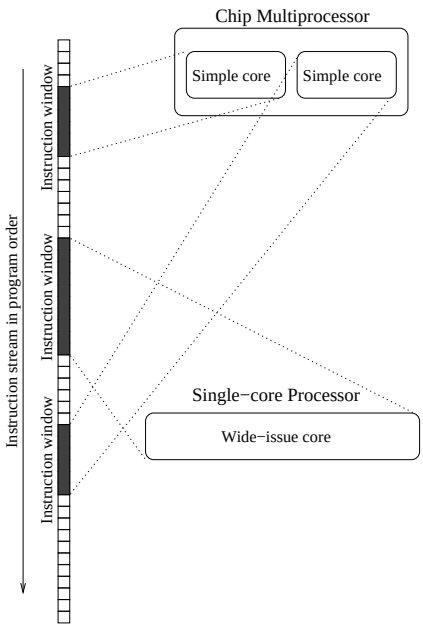


Figure 2.3: *Comparison of instruction windows for a wide-issue processor core and CMP with two simpler cores.*

The instruction window of the wide-issue single-core processor in Figure 2.3, i.e. a processor which can execute many instruction at the same time, is larger than the windows for the two simpler cores contained in the chip multiprocessor in the same figure. Therefore, the wide-issue core has a larger potential to exploit instruction-level parallelism than one of the simple cores in the CMP. However, it has been found that increasing the size of the instruction window over a certain point yields diminishing returns for exploiting ILP. In addition, processor cores become too complex and too slow if the window size is very large.

Parallelism limit studies have concluded that despite the diminishing returns for increasing the size of the instruction window, there is much parallelism in common applications [LW92, PGTM99, Wal91]. However, the parallelism exists between instructions much further apart in the instruction stream that current processors can exploit. Thread-level speculation can be seen as a way to increase the instruction window. Instead of expanding a single instruction window, there is a virtual window consisting of several smaller windows each maintained in a separate processor. If the chip multiprocessor in Figure 2.3 executes several speculative threads derived from a single application in this manner, parallelism between instructions very far apart can be exploited. Even if the instruction window for each core in the

CMP is relatively small, the combined virtual window is large.

The main difference between TLS and a traditional parallel program is that the speculative threads are not provably independent. Instead a run-time *speculation system* detects *dependence violations* and resolves them by *rolling back*, or restarting, threads as necessary. When a thread is restarted, the results produced by the thread must be thrown away, or *squashed*. The terms roll-back, restart and squash are used more or less interchangeably in the thesis. When a thread is restarted, it is implied that erroneous data is also squashed. If a thread is said to be squashed, it may or may not be restarted depending on the situation and restart policy used.

Since all threads are part of a sequential program, there is a natural order among them. The threads are ordered according to their relative position in a sequential program; a spawned thread is said to be more speculative than another thread if it would have executed after that thread in the sequential case, or less speculative if it would have executed before. If the results from these threads are *committed*, or merged with main memory state, in that same order, the end result is guaranteed to be correct.[3]

2.2.1 TLS Execution Model

The functionality required for TLS will be illustrated with an example. The example shows how to exploit module-level parallelism with thread-level speculation, though the same principles apply for other sources of parallelism. Pseudo-code for a short program is shown to the left in Figure 2.4. The main routine calls two functions, f1() and f2(). The f1() function returns a value, f2() does not.

The sequential thread view in the center of the figure shows how the execution of the main thread is interrupted for every function call. Horizontal lines in the thread represent jumps to another function, or corresponding returns, while vertical lines represent execution of a function. The function name is indicated at the bottom of each vertical line. The dotted line shows how f1() returns a value *b* to the main function.

Starting threads: In the TLS thread view the program is split into three threads. When execution starts, there is only a single thread, the non-speculative thread T1. When execution reaches the call to f1(), a new thread is spawned. The new speculative thread, T2, begins to execute the code after the call instruction, that is at the module continuation. Thread T2 is dependent on thread T1, i.e. results computed in T1 may be used by T2. The final thread, T3, is spawned from T2, and is also speculative. T3 may have dependences on both T2 and T1.

Dependence violations: The non-speculative thread will never be squashed since it has no earlier thread to be dependent on, thus forward progress is guaranteed

[3] Correct is defined as producing the exact same result as when executing the application sequentially.

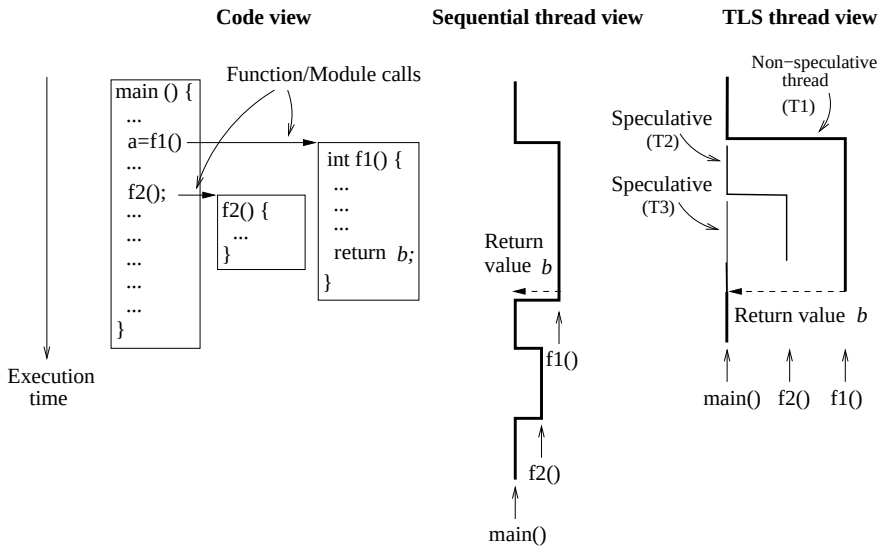


Figure 2.4: *TLS example: Code snippet with two function calls, sequential execution compared to module-level speculative threads.*

as long as the non-speculative thread is allowed to execute. Other threads, however, can suffer from dependence violations. For instance, if thread T2 in the example uses the return value from f1() immediately after the call, this read will cause a data dependence violation. As execution of T1 reaches the return statement, the speculation system must determine if T2 has used an erroneous value of *b* instead of the value to be returned. If that is the case, all threads that may have been affected by the erroneous value will have to be squashed and re-executed. As long as a thread may be affected by a dependence violation, all results it produces are speculative results and must be possible to undo. There are two types of dependences, data and control dependences. These dependences are described in more detail in the following two sections.

Committing threads: When a thread has finished executing and is known to not have any dependences on earlier threads, it may commit its results. To commit the results means that the speculative results are merged with non-speculative state and no longer have to be possible to undo. Since the TLS execution must produce the same end result as a sequential execution of the application, it is necessary that the threads commit in the order specified by sequential execution. Therefore a thread may not commit until it has become the non-speculative thread. In the example, when T1 has finished executing, the next thread in the sequential order, T2, becomes the new

non-speculative thread.

There is another reason why only the non-speculative thread may commit. The speculative threads T2 and T3 may have dependences on T1. Since these dependences could arise at run-time as a result of input-dependent computation in T1, it is not possible to know for certain that a speculative thread will not have to be squashed until all previous threads have finished executing and no dependences are detected. This happens when the thread becomes non-speculative

For these two reasons, threads can complete their last instruction in non-sequential order but they can not be committed and successfully retired from the speculation system until they become the non-speculative thread.

Data Dependences

Data dependences arise when instructions read and write to the same memory or register location. The basic intuitive model for all memory locations is that whenever data is read from a location, one expect to obtain the value that was most recently written to that same location. In a shared memory multiprocessor system, this principle extends to memory locations written by any processor in the system.

In a sequentially consistent system, all possible interleavings of memory accesses from the processors in the system are considered to be correct, as long as all accesses from each individual processor appears in program order. However, the total order of memory accesses in the system must appear to be the same for all processors. Program order is the order in which the instructions occur in the original code with respect to a single thread.

For TLS, is is not true that all possible interleavings are correct. Since the threads originate from one program there is only one single correct order for all data accesses that the parallelized version must obey, the program order of the sequential program. This means the result for all memory accesses in a TLS system must eventually appear to occur in the same order as for the sequential execution.

There are three types of data dependences: *flow-, anti-, and output dependences*. Figure 2.5 shows an example where all types are included. Each vertical box in the figure represent a thread containing instructions, but only read (R) and write (W) instructions are shown. For instance, *Ra* means read from location *a* and *Wb* means write to location *b*.

In the original thread to the left, the dependences shown with arrows (1) and (5) are anti-dependences. A write updates the location, which means a preceding read cannot be allowed to occur after the write. If that would happen, the read would return another value than specified by the program order. Arrows (2) and (6) show output-dependences, that is two writes that update the same location. If the order is not maintained, the location will contain the wrong value after the second write.

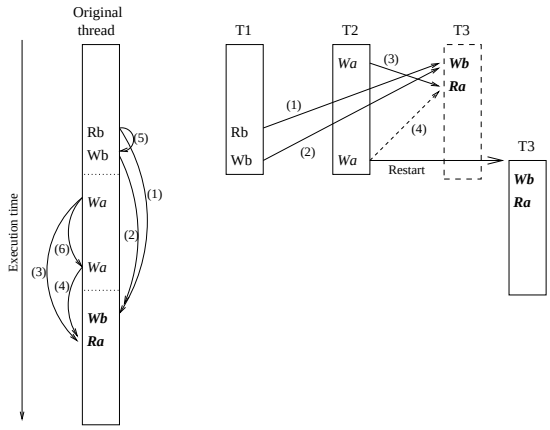


Figure 2.5: *The data dependence problem with thread-level speculation.*

Finally, (3) and (4) are flow, or true, dependences. When a location is updated and subsequently read, the read must occur after the write, otherwise it will not return the desired value.

When one of these dependences are not respected, we have a write-after-read (WAR), write-after-write (WAW) or read-after-write (RAW) hazard respectively [HP02]. On the right of Figure 2.5 the original thread has been split up into three new threads. The leftmost, T1, is non-speculative since it is the first thread. Between T1 and T2 thereare no dependences, since they do not access the same locations.

However, the *Wb* instruction in T3 depends on both *Rb* and *Wb* in T1. Hence, the anti-dependence has given rise to a WAR hazard (1) and the output dependence to a WAW hazard (2). Doing nothing to resolve this problem will mean that *Rb* in T1 does not get the right value, and the *Wb* in T1 creates an erroneous final value for location *b*. This happens since the write in T3 will occur earlier in time than both instructions from T1. Anti- and output dependences are also called *name dependences*, since they arise due to reuse of memory locations. These hazards can be avoided by removing the name reuse causing the hazard.[4]

The two flow dependences (3) and (4) between T2 and T3 will result in RAW hazards when the code is parallelized. Flow dependences cannot be resolved as easily as name dependences, since there is actually a new value communicated between the write in T2 and the read in T3. Arrow (3) points downwards, which means the new value is produced in T2 before it is consumed in T3. As long as the value is

[4]Register renaming is a technique that performs this separation of storage locations on-the-fly in order to overcome WAW and WAR hazards for registers in the processor.

propagated to the consuming instruction, this hazard can be avoided.

We can conclude that it is desirable for writes to propagate to more speculative threads in order to avoid RAW hazards, but undesirable that writes are visible for less speculative threads in order to avoid WAW and WAR hazards.

For the final RAW hazard (4), T2 has not performed the store, and likely not even computed the value to be communicated when T3 performs the read. Even if the value is propagated to thread T3, it will arrive too late. If the dependence is known in advance, for instance after compiler analysis, a possibility would be to stall T3 until the value has been propagated. As has been discussed, however, this is not always possible.

In a TLS system, T3 will speculatively read location *a*, betting that no such hazards will occur. However, the system must be able to detect a dependence violation if it happens; the violation can be detected when T2 performs the write. When the violation is detected, the speculation system will know that the value previously read by T3 was likely not the correct value, and the computations using this value will likely have produced incorrect results. After such a dependence violation, all computations that in any way derive from the incorrect value will need to be redone, and any results and other side-effects must be undone. This is a roll-back.

Control Dependences

Since speculative threads are started ahead of the non-speculative thread, they may be control speculative as well as data speculative. For instance, if loop iterations are used to spawn new speculative threads, it is not always possible to know how many iterations will be executed. In that case, future loop iterations can be started speculatively. If the loop exits and there are still speculative threads running for future iterations, this is a control dependence violation.

Threads started at module continuations, such as in the example in Figure 2.4, are usually not control speculative. If a function is called, execution will almost always return to the function continuation.[5]

After a control dependence violation, the threads on the wrong control path have to be squashed. This is similar to a roll-back due to a data dependence violation, except the code is not re-executed again since it should not have been executed in the first place. The superthreaded architecture [TY96] is an example of an architecture with speculative threads that does control speculation, but no data speculation.

Investigating the effects of control speculation is, however, beyond the scope of this thesis.

[5]There are some infrequent exceptions. If instructions such as setjmp/longjmp are used, it can cause control misspeculations with module-level threads.

Components of a Speculation System

With the example in mind one can conclude that the speculation system, which manages the speculative threads, should be able to perform the following basic functions:

- Select when to spawn threads, and handle thread starts.
- Detect dependence violations and be able to roll back execution to a known correct state.
- Commit speculative results and retire threads from the speculation system as they become non-speculative.

Avoiding dependence violations due to name dependences, as well as flow dependences when the value can be forwarded, is not a requirement for correctness. However, it is an integral part of most speculation systems for performance reasons. It is considered part of the basic functionality of a speculation system in this work.

The following sections will discuss the issues involved for implementing these basic TLS functions. The first part of this thesis will not be concerned with implementation details. Instead a full-featured speculation system is assumed. This survey of techniques is partly intended to make a plausible case for the feasibility of building the machine models used in the following chapters.

2.2.2 Thread Selection and Thread-Start

Any thread-level speculation system must have a way to divide sequential applications into threads. This includes both a policy dictating where to spawn off new threads, and a mechanism to get the new threads started on another processor.

Ideally, the threads should be independent, or at least not contain any dependences that cannot be resolved by the speculation system without resorting to a roll-back and loss of work. On the other hand, the whole point of TLS is the possibility to optimistically spawn a speculative thread even if it is unknown whether it is dependent on other threads or not. This means, for the sake of correctness, dependences are not a problem. However, roll-backs and thread-starts are operations that will come with some amount of overhead, so if misspeculations are too frequent, the performance will suffer. In addition, if there is no independent work to be performed in the speculative threads, the sought-after performance boost will fail to materialize.

Many TLS projects assume access to the source code of the sequential applications, and therefore the possibility of finding promising decompositions with compiler analysis. In this thesis, only access to the program binary is assumed, and thus only techniques that can be applied at run-time or possibly with some binary translation are considered. The survey of TLS architectures in Section 2.3 indicate which of these two categories the major TLS projects belong to.

The most common target when looking for promising threads is loops. In loop-level speculation new threads are spawned for loop iterations; successive iterations are run in parallel instead of sequentially. When a loop is encountered, many threads can potentially be spawned at once for many successive iterations. Only the second part of this thesis includes loop-level threads. The rationale behind loop-level threads is that loops often perform the same calculations over a set of data where each iteration is independent of the other, but if pointers are used it is difficult to exploit this parallelism with static methods such as parallelizing compilers.

Module-level speculation is used throughout this thesis and investigated in detail in the following chapters. It treats module calls (i.e. function, procedure or method invocations) as potential points where a new speculative thread can be spawned. When encountering a call instruction, the original thread will continue to execute the called function, while a new thread is created for execution of the module continuation, i.e. the code after the call instruction. The example in Figure 2.4 shows module-level threads being created as functions f1() and f2() were called.

Module-level parallelism has a number of advantages. It is easy to identify threads at run-time as they start at call instructions and end at return instructions. There is typically no control misspeculation; a called function will almost always return and the code after the function will then be executed. Functions, ideally, work mostly on local data, minimizing the risk for inter-thread dependences, except for the return value which is a common dependency. Module-level parallelism has also been investigated in the Hydra project [HWO98], by Chen and Olukotun [CO98], by Oplinger et al. [OHL99], and later by Hu et al. [HBJ02] and Renau et al. [RTL+05].

There are also a number of disadvantages with module-level parallelism. As opposed to loop-level threads, only one new thread can be created at a time; there is no common spawn point where a whole cluster of threads can be spawned, potentially making thread-start less efficient. It is also more challenging to keep track of the order of module-level threads. With loop threads, one can create a system where a new thread is always the most speculative thread. With module-level threads, this would be a severe restriction. On the contrary, for efficient exploitation of module-level parallelism, the speculation system should be able to spawn threads out-of-order. This means a new thread can be spawned even if there are both more and less speculative existing threads in the system. Note that even for loop-level threads out-of-order spawn is necessary if the speculation system is expected to be able to spawn threads from multiple levels of a nested loop.

It is important to keep track of the sequential order of the spawned threads. This order is used for dependence detection and in the commit phase. The order is defined by the original sequential application. When spawning a new thread it will be more speculative than its parent, but will inherit the relationship of its parent with respect to all other threads. As an example, Figure 2.6 extends Figure 2.4 with another function

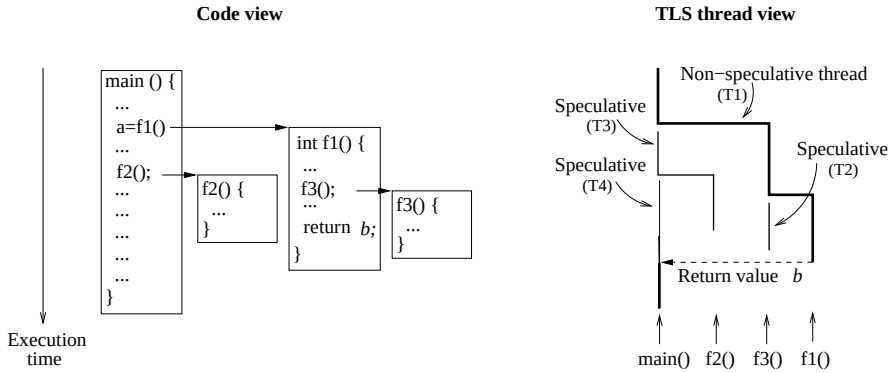


Figure 2.6: *TLS Example: Three function calls. Thread T2 is started out-of-order with respect to T3 and T4.*

call, more specifically a call to f3() from f1().

Since this call would be executed before the call f2() and the continuation for f1() in the sequential case, its thread order is lower than the threads created at the calls to f1() and f2(), even if both those threads were created earlier in time. Thus, the correct thread order is the one shown in the figure: the thread spawned for the call to f3() is T2, while the threads spawned for the f1() and f2() calls are T3 and T4 respectively.

Out-of-order spawn implementations are described by STAMPede [SCM97], Hydra [HWO98], DMT [AD98], and Renau et al. [RTL+05, RSC+05]. Hydra and DMT maintain thread order with a dynamic structure, where new threads can be inserted in any position. The other two proposals use sequence numbers but leave holes in the sequence where out-of-order threads can be inserted. Using a dynamic list or tree method is more flexible, but difficult to implement in an efficient manner for most architectures. Many other TLS systems only support in-order spawn.

In addition to loop- and module-level threads, several other schemes have been proposed. MEM-slicing [CW99b] creates threads based on memory accesses. A new thread is started dynamically at a load or store instruction. After a minimum amount of instructions have been executed and a new load/store occurs, a new thread is started again. Marcuello and González [MG02] use profile-based analysis to create threads based on three main criteria: high probability that execution reaches the spawned thread (few control misspeculations), few data dependences between threads (few data misspeculations) and thread size (not too long or short). Finally, Trace processors [RJSS97] take control flow into account when selecting traces. Trace creation is terminated when a call indirect, jump, or return instruction is encountered, or at most after 16 instructions. Several architectures let a compiler decide where to spawn threads. The compilers use various heuristics to find promising threads – compiler

techniques are discussed in Section 2.5.

The desired size of threads varies greatly between different architectures. For instance, Trace processors with tight coupling between the cores have a maximum of 16 instructions in a trace, while the Hydra project found that threads of 300-3000 instructions are preferable for their CMP architecture.

A new thread will get its initial state, including register contents and starting address, from the original thread. In various architectures, this is physically done either through the memory system, with a dedicated register bus, or if simultaneous multithreading is the base architecture, with a fast copy of values or register map. Examples of architectures using each of these techniques are given in Section 2.3.

2.2.3 Managing Speculative State

As long as a thread is speculative, it is possible that execution must roll back. Therefore, guaranteed to be correct results must not be irreversibly overwritten by speculative threads. Furthermore, in order to avoid name dependence violations it is necessary that the speculation system is able to maintain multiple versions of the same memory location. Finally, in order to detect dependence violations, maintaining a record of speculative memory accesses is necessary. The solutions for these problems are related, and therefore will be discussed together. All the information stored for the speculative threads is called the *speculative state*.

Most proposed TLS machines catch the results from speculative stores in either special-purpose buffers, or in a modified cache hierarchy, until the thread can commit its results. Steffan and Mowry have found [SM98] that for the small-grain threads usually considered for CMPs, the cache space seems sufficient to hold the speculative state.

At least one early proposal used a single centralized buffer [FS96]. However, the scalability of such a solution is limited. Therefore, most TLS architectures use some form of distributed storage.

The speculative versioning cache (SVC) by Gopal et al.[GVSS98] is one of many designs using the L1 data cache with an extended coherence protocol to manage versioning and buffering of speculative values. I will use this design as an example to highlight the issues involved in maintaining the speculative state.

Example: Base SVC

Figure 2.7 shows a cache line for a base SVC with associated control bits. For now, we assume the data field is a single word per cache line. In addition to the common store bit *s* and valid bit *v*, the SVC cache line has a load bit *l*. The load bit indicates that the cache line has been loaded by the currently running thread before the thread performed a store to this location. This is called an *exposed load*.

Tag	V	S	**L**	**Pointer**	Data

Figure 2.7: *A cache line for a base speculative versioning cache (SVC).*

Looking back at Figure 2.5, one can notice that the unavoidable RAW hazards occur only for exposed loads; these loads represent data transfer from one thread to another. Tagging the exposed loads with the *l* bit makes it possible to detect RAW violations; when a less speculative thread writes to an address, the coherence protocol for a cache containing a more speculative version can signal a violation if the tags match and the *l* bit is set. This also means it is necessary to propagate all stores to remote caches that contain more speculative versions of a cache line. The SVC cache line contains a pointer field which identifies the cache containing the next more speculative version of the line, if any. The fields added to the cache line for speculative versioning are shown in bold in the figure.

Figure 2.8 shows an example of dependence detection with base SVC; it is the same example as in Figure 2.5, which means the example contains a flow dependence violation. The caches below the threads show only the valid, load, store and tag fields of two cache lines from the L1 data caches associated with each thread. The subscripts in the *l* and *s* fields indicate the temporal order of the data accesses.

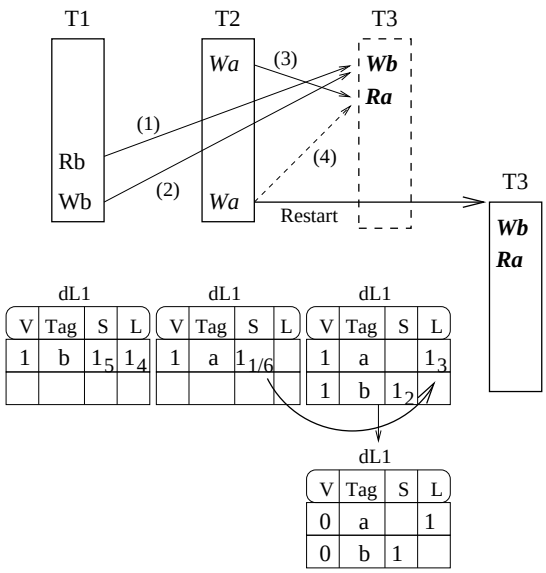


Figure 2.8: *TLS Example: Data dependence detection with base SVC L1 data caches.*

The first access is a write to location *a* in thread T2, which allocates a cache line in the cache of T2 and sets the valid and store bits. The second access is a write to *b* in T3. So far nothing out of the ordinary has occurred.

Access 3 is a read *a* from T3. This read should get the most recent update of this location, which was the write in T2. Therefore, the desired data is in the speculative cache line for location *a* held by T2. In the SVC, this is solved by the version control logic, which receives the state of the location from all caches and uses the next speculative version pointers to find the most recent version. The control logic compares versions and makes sure the requesting processor gets a response from the correct cache, or from main memory if the location has not been used by a previous thread. Thus, speculative writes are forwarded to more speculative threads.

Access 4, a read *a* from T1, does not get the updated value of *a* from T2 since it is more speculative than T1. Instead, T1 will read the value from main memory, and remain unaware of the write that has occurred in T2. This feature means the SVC will avoid name dependence violations.

All goes well until access 6, which is a write to *a* from T2. Since T3 is more speculative and has a copy of *a*, the write is propagated to its cache for dependence detection. The T3 cache detects that *a* is loaded with the *l* bit set, i.e. it is an exposed load. Therefore the cache controller signals a dependence violation to its corresponding processor.

When a dependence violation has occurred, the processor must undo all erroneous results created after the exposed load, and redo all computations needed to produce correct results. Theoretically, only instructions dependent of the exposed load causing the violation need to be re-executed. However, this is only possible if those exact instructions and results can be singled out.

SVC works at a larger granularity. If a thread suffers from a violation, all potentially erroneous threads, i.e. the thread in which the violation occurred and all more speculative threads, are squashed and restarted. The squash is performed by invalidating all cache lines with either the *s* or *l* bit set, since these may contain erroneous results, and then execution can be restarted from the first instruction in the thread. The state just after the roll-back is shown in the lower image of the T3 data cache.

There are some additional requirements for the SVC to work as intended. Cache lines with the *l* or *s* bit set can not be evicted from the cache and allowed to overwrite main memory until the thread is non-speculative. As mentioned, the results have to be committed in the correct order so that the end result is correct. In addition, evicting speculative cache lines would make dependence detection impossible; the speculation system needs the *s* and *l* bits of all speculative cache lines to detect flow dependence violations. Therefore, if a speculative thread cannot load a cache block due to lack of free non-speculative cache lines, the thread has to stall until it becomes non-speculative and can commit its speculative values.

If a thread finishes successfully and becomes non-speculative, all cache lines with an *s* bit, i.e. values that have been changed, must be written back to lower levels of the memory system before a new thread can use the processor. This is to prevent speculative state from the old and new thread from getting mixed up. There is no way to separate committed values from a committed thread from new speculative values in the cache.

Shortcomings of Base SVC

The base SVC highlights a number of potential limitations of a speculation system:

- Since the cache needs to write back all dirty cache lines when a thread is committed, there will be a burst of traffic for every commit. This will tie up the processor until the cache is clean, and also might slow down other threads.
- With SVC, the speculative state is tied to an L1 cache rather than the thread. This has several implications. First, a speculative thread can not migrate from one processor to another during execution. Worse, a new thread cannot use the processor until the preceding thread has committed, which means load-balancing becomes an issue. For loop-level speculation, where threads are usually of approximate equal length, this may not be a serious problem. However, module-level threads vary significantly in size. If a thread is much shorter than a less speculative thread, the processor executing the short thread may stall for a long time waiting for its turn to commit. Finally, for SMT cores where several threads share the L1 cache this scheme does not work as only one speculative thread at a time can store its state in the cache.
- Threads are assumed to be created in-order. That is, the most recently used thread is also the most speculative. As explained earlier, this presents a problem for module-level threads.
- If cache lines are more than one word wide, there will likely be false dependende violations, just as there is false sharing in a regular coherence protocol with long cache lines.
- A violation causes re-execution of the violating threads and all successive threads instead of just invalidating and re-executing code dependent on the read causing the violation. That means more re-execution than necessary.
- All speculatively loaded and modified cache lines are invalidated when a thread is squashed. Invalidating all speculative cache lines even if the values are correct means unnecessary cache misses when the thread is restarted.

In the plethora of proposed speculation systems, there are proposed designs targeting one or more of these limitations.

Proposed Designs for Managing Speculative State

While Figure 2.7 shows the minimal version of SVC, Figure 2.9 shows a more advanced version with performance improvements. Both versions are described by Gopal et al. [GVSS98].

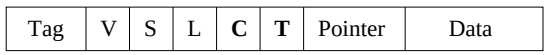


Figure 2.9: *A cache line for an optimized speculative versioning cache.*

This version solves the problem with bursty commit traffic. Base SVC has been extended with a commit bit *c* in the cache line. When a thread is to be committed, all lines with the *s* bit set also get the *c* bit set, and all *l* bits are reset. That way, committed and speculative values can be separated. Dirty cache lines are now written back lazily, just as in a regular write-back coherence protocol. This also means there may be several different committed versions at the same time, but only the most recently committed version should be used. Therefore, a *t* or stale bit is introduced to distinguish the most recent copy from stale copies. The SVC paper [GVSS98] also mentions how to solve the false violation problem simply by maintaining separate *s* and *l* bits for each word in the cache line. The number of *s* and *l* bits to use is a performance vs. cache overhead trade-off.

The STAMPede project also uses an enhanced coherence scheme [SCM97, SM98, SCZM00]. The major difference from SVC is that each processor/cache pair has a structure called *speculative context* which contains information for a speculative thread. Among other things, the structure contains an *epoch* or thread number, *l* and *s* bits for each cache line, and a list of all modified cache lines which may exist in other versions. This design has capabilities similar to the SVC. However, it can be easily augmented by adding several speculative contexts per processor, thereby enabling context switches, several speculative threads per cache (SMT), and avoiding stalling the processor when a thread is waiting for commit. Upon commit or squash, the list of speculative cache lines has to be traversed so that lines can be correctly committed or invalidated before the processor is reused. Migrating a thread to another processor is not possible in STAMPede.

The Hydra project [HWO98] uses another approach. This design employs a modified cache with dependence detection similar to SVC and special write buffers for speculative values. When a new thread is spawned, a free buffer is assigned to that thread. All speculative writes are stored in the buffer, which is kept for as long as

the thread is running. When a thread is committed, the speculative cache lines in the L1 cache can immediately be invalidated, and the contents of the buffer written back to the L2 cache. Squashing is done by invalidating the speculative values in the L1 cache and write buffer. This scheme has the advantage that the number of buffers can be greater than the number of processors. Therefore, the buffers can write back speculative values to the L2 on spare cycles, while the processor is immediately assigned a new buffer and set to work on a new thread. A drawback is, however, the need for additional buffers and the fact that the buffer size limits the amount of speculative state the thread can be allowed to produce.

Renau et al. [RSC+05] propose a more flexible design. Speculative cache lines from several threads can be mixed in the same cache; the blocks from different threads can be separated since they are tagged with a thread identification number. Committed blocks are written back to main memory lazily, as they are accessed. Committing or squashing threads is done by setting a commit or squash bit in the list of threads kept in each cache. There may be a need to clean out blocks in some cases when there are no free thread identification numbers left because lazy commit has not cleaned out all used blocks. This design allows for multiple threads per processor, thread migration, and avoids burst traffic at commit and squash.

Exceptions and I/O operations must be executed non-speculatively regardless of which of these systems is used. This because speculative threads are not allowed to alter the system state permanently in any way. I/O operations are, in general, difficult or impossible to undo. Also, it is possible that exceptions arise in the speculative threads that would never occur in sequential execution. For instance, a division with zero or segmentation fault can occur since the speculative thread has erroneous input values.

A taxonomy and survey of methods for buffering memory state is presented by Garzarán et al. [GPL+03]. In their taxonomy, the techniques are classified according to two criteria: separation of task state (versioning) and merging of task state (commit). The survey covers a few alternative ways to manage speculative state. For instance, instead of buffering speculative state and committing after the thread becomes non-speculative, main memory can be updated immediately if an undo log is kept [GPV+03, ZRT99].

Register-level Dependences

So far, only dependences through memory have been discussed. However, the locations in e.g. Figure 2.5 might as well be register values. When the application is parallelized, register locations with the same name get separated since each thread has its own register file. However, dependences through registers must be detected similarly to the memory references.

As opposed to memory dependences, register-level dependences can be found statically. For an architecture with compiler support, this makes it possible to avoid register-level dependence violations. The multiscalar architecture [SBV95] uses a unidirectional ring for register communication. Threads, or tasks in multiscalar vocabulary, are created statically by a compiler. The compiler also produces a *create mask* indicating which registers a task may produce and special operate-and-forward instructions that will send out the results on the communication ring. Subsequent tasks will wait on register values in previous tasks' create masks. A task can proceed when it has received the necessary register contents from earlier tasks.

Krishnan and Torrellas [KT99] use a related technique for a chip multiprocessor. Sequential applications are annotated with a binary annotator, obviating the need for source code. The annotations, together with some clever logic and a dedicated register bus, is used to communicate register values. If a register is not yet available, the consumer thread stalls, just as in a multiscalar processor.

Most TLS architectures do not propose dedicated register communication hardware. Instead, register values are communicated through memory. For instance, Hydra allocates a *register passing buffer* from a pool of buffers for each new thread. The buffer is filled with the new thread's initial register values and kept during the lifetime of the thread. Thus, if the thread is restarted, the initial registers are readily available. For module-level threads, the return value is predicted at thread-start and the prediction is validated when the thread becomes non-speculative. For loop-level threads, variables that may carry dependences across iterations can not be register allocated since no dependence detection mechanism exists for these registers.

In the STAMPede CMP [SCM97] the compiler inserts synchronization for registers and handles forwarding through a shared cache. That is, register dependences are synchronized.

A technique that can be used to detect register-level dependences using unmodified binaries is to save the initial register contents for the thread, and compare this to the final register contents of the previous thread when it finishes [CW99a, OL02]. If the values differ, there has been a misspeculation. This verification can begin as soon as a thread becomes non-speculative, i.e. when a thread commits the final register values can be compared to the next thread, even if that thread is still running. That way, verification can overlap execution in many cases.

In Speculative Multithreaded processors [MGT98], value prediction is used for live-in registers which have shown to be predictable, while other live-in registers are synchronized between the tightly-coupled processors, or thread units, using a special live-in register file.

Data Value Speculation

A technique that is often used in conjunction with data dependence speculation in order to reduce the number of roll-backs is data value speculation.

Even if there is a flow dependence violation, a roll-back is only necessary if the value that was used by the violating read is not the same as the one produced by the store instruction it is dependent upon. In some cases, that value can be predicted. If the speculation system supports data value prediction, roll-backs can be avoided for flow dependences when the prediction is correct. This also requires that the speculation system can check whether the prediction was correct when the value is finally produced. The predictability of data values has been investigated among others by Lipasti et al. [LS96, LWS96], and Sazeides and Smith [SS97].

In fact, the basic operation of TLS includes a form of prediction; the read predicts that the value which is currently found in the requested location is correct, i.e. last-value prediction. Even if a store occurs to the location at a later time, the read value might still be correct. Sometimes a store instruction will write the same value that was already stored in the location, a so called silent store [LL00]. However, most architectures lack the ability to detect that this happened and must pessimistically roll back for all flow dependences where the value could not be forwarded in time. Silent store elimination has been found to be a useful addition to TLS systems [SCZM02].

It has already been mentioned that some architectures try to predict the live-in values of registers, but it may also be applied to memory accesses. Value prediction has been investigated in a number of TLS projects [CW99a, HBJ02, MGT98, MTG99, OHL99, RJSS97].

2.2.4 Speculation System Summary

The previous sections have discussed a number of events where the speculation system intervenes with normal execution. These events are summarized in Table 2.1. While the design space allows for a number of different implementations and trade-offs, the result for each event described in the table reflects the functionality assumed in the baseline simulation models in this thesis.

2.3 TLS Architectures

Some TLS architectures have already been introduced; most of these designs were organized as extensions to a baseline CMP architecture. However, there are many other TLS projects that deserve attention.

The first mention of speculative threads, to the best of my knowledge, is in a paper by Tom Knight. Knight's 1986 paper [Kni86] describes hardware support for specu-

Table 2.1: *Speculation system events*

Event	*Result*	*Required functionality*
Thread-start	Start new speculative thread	Supply initial value (registers/PC etc)
Load	Supply the most recent version (speculative or non-speculative)	Version order tracked; updated values forwarded between tasks
Store	Store a value affecting this and subsequent threads	Multiple versions of same address can be managed; forwarding between tasks
Commit	Merge speculative state with safe memory state.	Merge speculative data with memory so that only the most recent (in program order) values remain
Violation detection	Detect if more speculative thread has a flow-dependency with less speculative thread	Sequentially defined order of speculative versions known
Roll-back	Invalidate speculative state due to dependence violation and and (optionally) restart thread	Ability to find dependent thread(s), invalidate its speculative state and reset initial values

lative threads on a multiprocessor, targeting programs in mostly functional languages (i.e. Multi-Lisp). The basic support needed for TLS was mentioned in Knight's paper. However, TLS research did not get off the ground for almost another decade, until the multiscalar project was initiated. Multiscalar processors [SBV95] have tightly coupled processing units in a ring configuration and supports thread-level speculation with a combination of hardware and compiler support.

During the years following the multiscalar project, a multitude of TLS architectures have been proposed. The work presented in this thesis is most closely related to architectures extending chip multiprocessors and simultaneous multithreaded processors, but TLS with other base architectures has been investigated as well.

This section will summarize the existing proposals for TLS categorized by their base architecture: tightly-coupled cores, shared-memory multiprocessors, chip multiprocessors, simultaneous multithreaded processors, and finally software-only solutions. For each proposal, key differentiating implementation choices and the functionality of the speculation support is summarized. The summaries also include some comments on the impact of overhead and whether the techniques are dynamic or re-

quire recompilation. These properties are highlighted for the purpose of comparison with the work in this thesis.

The section is quite lengthy; however, it can be skipped without reduced understanding of the following chapters as the key concepts and techniques have already been introduced.

One TLS proposal that should be mentioned but did not fit in any of the categories below is **SPSM** [DOO+95] (1995). It is an early TLS architecture, but without a specific target machine model.

2.3.1 Tightly Coupled TLS Architectures

Tightly coupled TLS architectures have multiple processor cores, but the cores are closely integrated. Typically, this means direct communication between the register files. These architectures are tuned to TLS, as opposed to the other TLS architectures which are extensions to designs originally intended for general-purpose multiprocessing.

Multiscalar processors [SBV95] (1995) have already been mentioned. In fact, the basic concepts behind multiscalar processors were introduced even earlier as the expandable split window paradigm [FS92] (1992). Multiscalar processors are based around a number of processing units, connected by a unidirectional ring for register communication. The architecture relies on the compiler defining tasks to run on the units. A task can only start a single successor task, and tasks will tie up their processing unit until it is designated the head task and can commit. The original multiscalar architecture uses a hardware structure called the address resolution buffer (ARB) [FS96] to store speculative results and check for dependences. The ARB checks for dependences in a manner similar to the SVC, but is a centralized structure. The main drawback with this solution is that is does not scale; memory accesses from all processors must be handled by the ARB. In fact, the SVC [GVSS98] is a solution for this problem from the same research group. In another follow-up paper by Vijaykumar [VS98], the main sources of overhead are identified as control flow speculation, data communication, data dependence speculation, load imbalance, and task overhead. Compiler techniques to reduce some of these overheads were also investigated.

The superthreaded architecture [TY96] (1996) uses an architecture similar to multiscalar processors. A number of cores are connected with a unidirectional ring. However, the superthreaded architecture does not speculate on data dependences, only control dependences. Execution of a thread is divided into several parts. The continuation stage takes care of forking the next thread. Next, target store address calculations are hoisted to the beginning of the thread, and the result forwarded to the successor thread. When this is done, the successor may start its own address com-

putations; knowing all possible dependences the successor will stall in cases where there is a RAW hazard. Still, a thread may be the victim of a control depencence violation. Therefore, this architecture also needs to store speculative results until the thread becomes non-speculative; speculative accesses are stored in a special buffer connected to each core.

Trace processors [RJSS97] (1997) is an architecture built around instruction traces. Traces are dynamic instruction sequences captured during execution and stored sequentially in a trace cache regardless of their original position in the code. Instead of branch prediction, the processor does next-trace prediction. The trace processor has a number of processing units with their own execution units and local registers, as well as a copy of global registers. Traces are executed speculatively; live-in register values are predicted, and dependence speculation used for memory accesses. Since trace processors work with parallelism at a finer granularity than most other TLS architectues, the squash/restart mechanism differs. The trace processor can restart a trace with the new correct value after a misspeculation and reissue only instructions along that location's dependence chain. The same mechanism that a regular out-of-order processor employs for recovery after misprediction is used to achieve this. Therefore, less extra work is required compared to the coarse-grained squashing most architectures must resort to in order to reduce the complexity of restarts. Speculative values are retained in a store buffer where multiple versions are separated with a sequence number, though the exact implementation of this buffer is not discussed.

Speculative multithreaded (SM) processors [MGT98, MG99a] (1998) also use a number of processing units, called thread units, interconnected in a ring topology. This architecture is specialized for loops. The loops are detected at run-time, no modification of the code or assistance from the compiler is needed. Several iterations run in parallel on the thread units. In the base architecture fetch bandwidth is conserved by sharing the fetch unit. The single fetched instruction stream is broadcast to all units. This works as long as the units are all on the same control path; if a unit takes another path, the thread and all successors are squashed. For integer codes, identified problems are that iterations often do not have the same control flow, and also that the number of iterations is typically low, which means reduced potential as the architecture can only speculate on one loop at a time. A more advanced version of the architecture handles multiple control flows.

The functional units of one thread unit are connected to a live-in register file in the successor unit. Input register values can either be predicted or synchronized with the previous thread. Value prediction is implemented with the aid of history-based information stored in a loop iteration table. Memory versioning is handled with the multi-value cache. The multi-value cache can hold a separate version of a memory location for each available thread unit. As opposed to other TLS proposals, SM tries to predict the addresses of stores, and the value of live-in registers. Store

addresses are broadcast to successor thread units that check their load/store queue for corresponding reads. If a dependence violation is detected, the violating and successor threads are squashed.

Pinot [OTKM05] (2005) is recent work on an architecture with some similarities with multiscalar processors. Pinot uses a unidirectional ring for register communication, but in contrast to multiscalar does not synchronize register values. A versioning cache for storing speculative state. A unique feature of Pinit is the binary translation tool which extracts speculative threads at a coarser grain than the multiscalar compiler. Like multiscalar, a thread can only have one successor, and threads are spawned in-order.

2.3.2 Chip Multiprocessor TLS

This category contains architectures that add TLS support on top of a chip multiprocessor. These architectures are typically intended to be useful both for multiprogrammed workloads and TLS, i.e. they are less specialized than the tightly coupled architectures.

Oplinger et al. [OHL+97] (1997) describe a CMP architecture for thread-level speculation on loops. The threads are specified by a compiler; the compiler also inserts synchronization operations to delay reads and reduce the number of misspeculations. A main thread executes the application, and signals dormant slave threads to start executing a loop when speculation is activated. The hardware consists of extra bits in the L1 cache to mark speculative data, as well as write buffers; there are twice as many buffers as processor cores for double buffering. Speculative data is thus stored both in the L1 cache and in the buffers. At commit, the L1 cache can be cleaned immediately and the processor recycled using the empty buffer while the full buffer is flushed into architectural main memory in the background. Therefore, commits create bursts of traffic, but will not prevent a new thread from starting in the meantime. For a squash, the data in the buffer is invalidated.

The Hydra project [ONH+96] (1996) was one of the first research projects proposing chip multiprocessors. The architecture was later used as a substrate for TLS [HWO98] (1998). The Hydra TLS project targets loop- and module-level parallelism. The architecture is a CMP with extra buffers for storing speculative state. One buffer is assigned for each speculative thread, and there is a separate bus for speculative stores. The speculation system is implemented as interrupt handlers executed by a special speculation control unit. The implementation is detailed, with measurements for the overhead of thread management. These figures have been used as one important, though not the only, data point on thread-management overhead when conducting experiments in this thesis. They note that the impact of frequent squashes and restarts, as well as the software control overhead, can be significant

and kill the potential speedup. Their attempt at exploiting module parallelism did not work well due to control overhead and misprediction penalties.

The STAMPede CMP [SCM97, SM98] (1998) is a proposal described in great detail. Register dependences and some data dependences are handled with synchronization, which requires compiler support. Compiler support is also assumed for choosing threads, optimizing them for speculation, and inserts calls to the speculation system, which is largely software-based. Hardware is, as mentioned earlier in this chapter, used for buffering speculative state, detecting misspeculations, and committing or squashing results.

Krishnan and Torrellas [KT98] (1998) propose an architecture where binary annotation of a sequential program is enough to create speculative threads. In addition, this is the only CMP model incorporating a bus for direct communication between the register files. Register-level dependences are synchronized using the synchronizing scoreboard and communication via the dedicated bus. Memory-level dependences are detected using the memory disambiguation table (MDT), which is a central resource in the CMP. It works much like the SVC, except the load and store bits for all cores are kept in the central MDT. This scheme is more like a directory-based protocol and does not need a snoopy-based coherence protocol. The L1 caches contain information used to reduce the load on the MDT; thus the MDT is not accessed for every memory operation. Speculative writes are stored in the private L1 caches and cannot be written back until the thread is non-speculative. The non-speculative thread works in write-through mode. However, any dirty cache lines which remain when the thread commits must be flushed back before the core is recycled for a new thread. Therefore, this scheme may cause traffic bursts at commit.

Atlas chip multiprocessor [CW99a] (1999) uses a novel algorithm, MEM-slicing, to create threads. In addition, it relies on an advanced value predictor to resolve inter-thread dependences. Everything is done dynamically, on sequential binaries. A novel feature is the dependence predictor for memory values, something few architectures have attempted. Atlas uses a hybrid correlating value predictor. The predictor does not contain predictions for all memory locations. Instead, a history-based dependency predictor is used to select which addresses to use value prediction for. The roll-back mechanism borrows from DMT, described below.

MAJC [Tre99] was an attempt to build a new CMP architecture, especially geared towards efficient execution of Java applications. It also incorporated space-time computing, which was a form of TLS for Java, and builds heavily on software support. This is the only real architecture that has so far included TLS. Unfortunately, the MAJC architecture was abandoned and space-time computing never reached the marketplace.

Multiplex [OKP+01] (2001) is an attempt to create an architecture that is efficient for both thread-level speculation (implicit parallelism in the authors' parlance) and

explicit parallelism. The Multiplex compiler partitions a program into explicit threads whenever possible, and when that fails creates implicit threads. The hardware can switch between threading modes, but cannot execute explicit and implicit threads simultaneously. The cache coherence protocol is derived from SVC, but modified so that the protocol can double as a traditional write-invalidate protocol for the explicit threads. Register dependences are synchronized.

Renau et al. [RTL+05, RSC+05] (2005) describe a flexible architecture with out-of-order spawn supporting both loop- and module-level parallelism. The CMP consists of a number of cores connected in a ring; this architecture is chosen to simplify the coherence protocol. Each processor has an array of thread, or task structures that makes it possible for several speculative threads to share a cache. They use a profiling compiler to define speculative tasks, but the hardware can dynamically merge tasks to reduce overhead and power consumption.

2.3.3 Multithreaded Processor TLS

Thread-level speculation for multithreaded processors seems like a good match due to the possibility for fast mechanisms to start new threads; initial data is already present in the same core and communication can be even faster than with the CMP.

Dynamic Multithreading (DMT) [AD98] (1998) is a TLS system based on simultaneous multithreading. Threads are handled purely in hardware with unmodified sequential binaries. Threads are created for loop iterations and module continuations. Instructions are stored in per-thread trace buffers after being fetched and decoded; instructions are retained in the buffers until the thread is non-speculative. Memory operations are similarly kept in the load/store queues until the thread commits. At commit, the load entries are freed and the stores performed. Also, the predicted live-in registers are compared to the real values, and loads are disambiguated in the load queue. If there is a misprediction the instructions are reissued from the trace buffer. Only affected instructions need to be executed again. The input values to new threads are copied from parent thread to child thread through a fast in-core copy mechanism. Thus, the thread startup is relatively fast. The roll-back mechanism has been criticized for being difficult to implement efficiently [PV03].

Marcuello and González [MG99b] present the DaSM architecture which is similar to their speculative multithreadeded (SM) architecture but on an SMT substrate. Each thread has a unique register map but shares the physical register file. When a new thread is spawned, a new register is allocated for all live registers, and they are, unlike DMT, initiated with a predicted value. Also unlike DMT, speculative values are stored in the first-level cache, not buffered in the load/store queues. This introduces some additional complexity but enables the use of larger threads. Like the SM architecture, DaSM takes advantage of the fact that different iterations in a loop

execute the same code, and can share the fetch bandwidth.

Implicitly-Multithreaded Processors (IMT) [PV03] (2003) lets the compiler select where to spawn threads. In fact, the same compiler is used as in the multiscalar project. Like DMT, IMT uses the load/store queues for memory disambiguation, and like multiscalar, threads are spawned in program order. Novelties in this architecture are a fetch-policy that is resource- and dependence-based, a technique for overlapping thread-start with execution to hide overhead, and multiplexing several thread contexts onto each executing thread supported by the base SMT architecture. This last feature reduces the load-balancing problem, however, compared to other architectures it does so in a hardware-efficient way and makes it possible for the multiple threads multiplexed onto the SMT threads execute simultaneously. Register dependences are synchronized by extensions to the rename logic; the creation of a rename map for the new thread is started ahead of thread spawn to hide the startup latency. This efficiency enables IMT to mine parallelism from very short threads.

2.3.4 Shared-Memory Multiprocessor TLS

Shared-memory multiprocessor architectures have been around for a long time compared to the previously discussed architectures. They were built long before integrating multithreading on a single chip was made possible by the growing transistor count on a single die. A number of TLS architectures retrofitting TLS for shared memory multiprocessors have been proposed. Typical for these architectures is that the inter-processor communication latencies are much higher than for the previously mentioned designs, which also means the threads need to be more coarse-grained in order to amortize the communication overhead. These architectures are less connected with this thesis as the focus is on-chip multithreading.

Knight [Kni86] (1986) describes an architecture targeted at mostly functional languages. The program is divided into blocks with a sequence number. The hardware is a shared-memory multiprocessor, where each processor has two caches. The dependency cache contains speculatively loaded values, and the confirm cache holds speculatively written data. The dependency cache snoops the bus for writes from other processors; if a write is to a location which is marked *depend* the computation in the processor is restarted or aborted. A block counter keeps track of the non-speculative block. If the counter reaches a block and it has finished executing the block can commit, which involves writing back the data in the confirm cache. In short, this early architecture contains basically the same elements as later TLS architectures. No performance evaluation is done, however.

The superthreaded architecture [KL98] (1998) was originally a tightly coupled architecture. The basic techniques behind the superthreaded architecture were adapted to shared-memory multiprocessors by Kazi and Lilja.

Zhang et al. [ZRT98, ZRT99] (1998) present an architecture where dependence detection is combined with the cache coherence protocol for distributed shared-memory multiprocessors. This machine is used to parallelize loops. The size of speculative threads is not limited by storage in this architecture.

STAMPede [SCZM00] (2000) is another project originally for chip multiprocessors, but later the system was evaluated for conventional multiprocessors as well.

Cintra et al. [CMT00] (2000) describe an architecture that enables TLS for shared-memory multiprocessors where each node is a speculative CMP. As opposed to Zhang and STAMPede, this architecture is hierarchical which makes it more flexible regarding the architecture of the processors used at the nodes. A cluster of threads are assigned to each node, but treated as a single thread by the speculation system. This way, the node may execute the threads in its cluster speculatively without affecting the upper-level speculation system. A global memory disambiguation table [KT99] is used to manage speculative state; the table is coupled with the directory in a CC-NUMA machine.

Prvulovic et al. [PGRT01] (2001) describes a system with some improvements compared to earlier systems: lazy commit, a speculative buffer overflow area, and reducing speculation-induced traffic improves the scalability of TLS on multiprocessors.

2.3.5 Software-only TLS

Finally, TLS can be implemented without hardware extensions. There are a number of proposals for software-driven TLS. Compared to hardware assisted TLS the overheads are higher. These techniques require access to source code for recompilation. The major advantage is the ability to use TLS on a regular multiprocessor machine. These techniques will only be described briefly since they are less closely related to the work in this thesis.

The LRPD test [RP95, RP99] (1995) is a run-time test to determine if there were any cross-iteration dependences for a loop that was speculatively executed in parallel as a do-all loop. Dependence violations are detected after the speculative threads have terminated, making misspeculations costly.

Kazi and Lilja [KL98] (1998) present another software scheme which employs dynamic renaming and synchronization of flow dependences.

Rundberg and Stenstrom [RS01] (2001) present an all-software speculation system that inserts highly tuned checking code for data accesses that a parallelizing compiler cannot disambiguate. The technique avoids name and some flow dependences with dynamic renaming and forwarding, and supports parallel commit for improved performance.

Cintra and Llanos [CL03, CL05] (2003) present a software speculation scheme based on sliding windows, and efficient data structures for supporting speculation operation. In this scheme, a fixed number of *chunks* can be scheduled at a time. A chunk consist of a small number of loop iterations. This scheme is found to reduce load imbalance problems and memory overhead associated with versioning compared to other schemes.

2.4 Transactional Memory

Many research groups have recently turned towards transactional memory. A transaction consists of a number of reads and writes which are commited at the same time. That is, a transaction provides failure atomicity, it must either be completed in its entirety or not at all. Transactions can be seen as a more general form of TLS, as there is not a predefined total order among transactions that reflect the program order in a sequential program.

For instance, **TCC** [HWC+04] (2004) is one of many proposals in the new wave of transactional memory based systems. The transactional memory coherence and consistency model replaces the traditional coherence/consistency models with atomic transactions as the basic unit of parallelism. A transaction is committed with a burst of writes after it has finished computing. Coherence is managed at the granularity of transactions instead of single memory accesses. Hardware-wise, the proposed TCC implementation resembles a TLS architecture; transactions must be able to commit or be squashed atomically, like a TLS thread. Transactions are more general than TLS threads, as there is no default order among commits. However, the programmer (or tool) has the option to specify the desired order.

Herlihy and Moss [HM93], Rajwar et al. [RG03, RHL05], Ananian et al. [AAK+05], McDonald et al. [MCC+05], and Moore et al. [MBM+06] among others have also been investigating the use of transactional memory.

2.5 Compiler Support for TLS

In this work, only techniques working on sequential binaries compiled with standard compilers are considered. Making use of compiler transformations specifically geared towards improving performance for a TLS system is beyond the scope of this thesis. However, given sequential source code, there are many compiler techniques that are useful for improving TLS performance. This section summarizes work on compiler techniques for TLS.

First of all, many techniques in parallelizing compilers, such as SUIF [HAA+96] or Polaris [BDE+96] could be useful for thread-level speculation as well; the work

carried out by a parallelizing compiler can be leveraged to reduce the number of dependences or synchronize known dependences. In fact, many TLS projects have done this.

Franklin and Sohi [FS92] discuss some possibilities for a compiler to rearrange code within a task/thread to improve performance. For instance, some statically detectable dependences may be avoided by hoisting computation of results to the beginning of a thread and pushing consuming instructions as far down in the subsequent thread as possible, thus increasing the parallel overlap of the threads. The compiler could also assist in partitioning the program into threads, which are super-sets of basic blocks and of appropriate size. These ideas are not implemented in the paper, but their potential is proved by manually arranging the code according to these principles.

Li et al. [LTW+96] show several techniques that can be useful for many TLS architectures: Variable privatization to reduce buffer overflow, last-write identification for data forwarding, workload analysis to assist thread partitioning, and reducing loop-carried data dependenced for improving loop speculation.

Oplinger et al. [OHL+97] discuss similar optimizations. Data from simulations of an ideal machine is used as feedback to the compiler to determine when to speculate and where to insert synchronizations. The algorithms are implemented in the SUIF compiler.

Vijaykumar and Sohi [VS98] present compiler techniques for selecting good tasks for multiscalar processors. These techniques are aimed at reducing inter-task data dependences and control flow, and to make sure tasks are not too short to amortize the start overhead.

Tsai et al. [TJY99] use compiler techniques to create threads for the superthreaded architecture. In order to improve performance from the baseline algorithm. Some of the techniques used include adjusting the thread size to avoid overflowing speculation buffers, using order-independent write operations in critical sections (which also requires hardware support), and converting data speculation (which the superthreaded architecture does not support) to control speculation.

Zhai et al. [ZCSM02] present compiler algorithms aimed at reducing the critical forwarding path; the compiler also inserts synchronizations where dependences can be found statically. They use the fact that the speculation system will guarantee correctness by using profiling and optimizing the code for the most frequently executed paths. In the infrequent cases where other paths are taken, this is allowed to result in a roll-back. In follow-up work [ZCSM04], they evaluate identification and synchronization of memory-resident value communications between threads.

The Jrpm system [CO03] is a Java run-time system and JIT compiler which dynamically parallelizes Java programs on a TLS architecture. The system uses runtime profiling and recompilation to create speculative threads. The compiler performs some optimizations to improve TLS performance such as synchronizing threads and

eliminating violations from some induction and reduction variables.

The Mitosis compiler [QMS+05] creates threads for the Mitosis architecture; the unique feature of this architecture is the use of precomputation slices at the beginning of each speculative thread. The precomputation slice computes the input values for the speculative thread using a heavily optimized version of the code leading up to the thread, hence reducing the risk for misspeculations. Since the speculation system will guarantee correctness, the compiler may use very aggressive and even unsafe optimizations to the code in the precomputation slice. The compiler creates the slices and also selects where to spawn threads.

Du et al. [DLL+04] use dependence profiling and a misspeculation cost model to determine where to spawn threads. Loop unrolling and software value prediction is also used.

Dou and Cintra [DC04] present a TLS compiler framework where expected run-length of threads, scheduling restrictions of the TLS system, and thread management overheads can be used to help the compiler find promising thread spawn points.

The POSH compiler [LTS+06] creates loop- and module-level threads. In addition, it uses profiling in order to filter out threads that are unlikely to improve performance. The compiler also tries to start the threads as early as possible, i.e. hoist the spawn point, to increase the parallelism. The hoisting distance, thread size, squash frequency, and prefetching benefits of threads are taken into account when deciding where to spawn threads. Finally, software value prediction is inserted at select places.

This chapter is an extended version of the previously published paper "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms" [WS01].

3

Limits on Module-Level Parallelism

The most popular form of speculative thread-level parallelism to exploit has been loop-level parallelism. While impressive parallelism can be obtained in numeric applications with loops that contain few loop-carried dependences, the poor parallelism coverage or lack of do-all loops in general integer applications severely limit this approach [OHL99]. On the other hand, module-level parallelism, i.e., parallelism across function, procedure, or method invocations, is potentially a more general and useful form of parallelism. First, it is very simple to identify the thread boundaries; new threads are created at module invocations, and terminated when they reach a return. Second, the control dependence problem encountered in, for instance, loop-level speculation is avoided. Presumably most importantly, however, is that modules are used frequently as the key abstraction mechanism in object-oriented programs in particular but also in imperative programming styles.

The first goal of this chapter is to understand to what extent the programming style – imperative versus object-oriented – affects the *inherent* speculative module-level parallelism. This is done by considering a set of C and Java programs and carrying out a speedup limit study assuming an idealized machine model. This model has an infinite number of processors, it supports perfect value prediction on return as well as memory values, and imposes no overhead for thread management or inter-thread communication.

The most important result gained from the experiments on the idealized model

is that there is a fair amount of module-level parallelism in C and Java programs. The question is how to best exploit it in terms of appropriate architectural support. I separate out a number of concerns through a series of successively refined architectural models as follows. The first issue studied is to what extent data dependences between threads limit the speedup obtained. More effective encapsulation of data in objects would speak in favor of an object-oriented style of programming. It is interesting to see whether this indeed will result in fewer data dependences and higher speedup limits when comparing C and Java programs. Another motivation is to see whether simple value prediction schemes suffice or research into more sophisticated value prediction schemes are warranted.

The next issue addressed is how much machine resources are needed to exploit the inherent parallelism. I address this issue by studying the speedup limit as a function of the number of processors, and also to which extent the number of available thread contexts, i.e. the number of threads that the speculation system can handle concurrently, affect the speedup.

I also investigate how thread management overheads impact on the achievable speedup to see whether research into more effective support is warranted and what this support should target. One interesting aspect is how well the granularity of parallelism in terms of common module sizes matches the overheads incurred in recent CMP proposals. Finally, the impact of the roll-back policy is considered. The impact of a more coarse-grained, but easier to implement, policy where threads start over from the beginning is compared to a fine-grained policy where threads can be restarted at the misspeculating instruction.

While [OHL99, OHW99, CO98] have also studied the potential of module-level parallelism in C and Java programs on CMP platforms, none of them has explicitly compared the nature of the module-level parallelism inherent in C and Java programs.

The main contributions are the insights into the inherent and architectural limits on the speedup for imperative versus object-oriented programs in a single consistent framework. The most important findings are:

- Overall, no significant qualitative differences between C and Java programs were found, suggesting that the programming style has a minor effect on the amount of parallelism to be exploited.

- The inherent module-level parallelism in applications is typically not more than four to eight. A small-scale CMP is enough to exploit the available parallelism provided more thread contexts than processors are available.

- Most of the codes do not benefit significantly from more advanced return value-prediction schemes than stride and last-value prediction suggesting that current predictors fare pretty well.

- The granularity of modules typically don't match the overheads in CMP proposals, e.g. Hydra.
- Dependences through memory is a major performance inhibitor, which suggests that more research into memory-value prediction schemes is warranted.

The architectural models used for the described experiments are presented in Section 3.1 and the methodology used for the simulations in Section 3.2. The experimental results are provided in Section 3.4. The work is put in perspective of related work in Section 3.5 before I conclude in Section 3.6.

3.1 Architectural Models

The models gradually introduce more of the architectural limitations associated with recent CMP proposals. For each new model, the additional limitations are specified and the insights one can gain are discussed.

Model 1: Inherent module-level parallelism

A speculative thread will successfully terminate as long as no data dependences are violated with threads that would precede it according to sequential semantics. Disregarding data dependences, the upper bound on the speedup is dictated by the control dependences between subsequent module invocations.

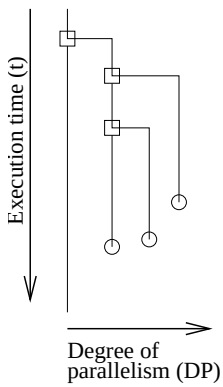


Figure 3.1: *Degree of parallelism in module-level speculation.*

Figure 3.1 shows an example of module-level parallelism: boxes mark module invocations (possible thread spawn points), circles show module returns (thread completion points), and the length of the vertical lines represents the relative execution time of the modules.

The degree of parallelism at any point in the execution, DP(t), equals the number of simultaneously running threads, which is affected by the amount and length of modules used in the application, as well as when they are called. In the figure, DP(t) varies from one to four as new threads start or are completed.

The first goal is to understand the scalability of module-level parallelism. To that end, a machine with an infinite number of processors is used. In this machine, no data dependences will be violated and cause roll-backs. That is, perfect value prediction for both register and memory accesses is assumed. In addition, thread management and inter-thread communication costs are zero.

This model provides important insights into the difference of imperative and object-oriented programming styles. One hypothesis is that an object-oriented style of programming would lead to more scalability in exploiting module-level parallelism, as the style encourages frequent use of methods when constructing programs. This is one of the hypotheses I will test using this model.

Model 2: Impact of data dependences

With this model, I am interested in investigating how data dependences between threads limit the achievable speedup and whether proposed support in terms of forwarding and value prediction in the recent literature is enough.

There are two classes of data dependences: flow and name dependences. In this as well as in the subsequent models, I assume that name (anti- and output) dependences can be resolved through renaming. On the other hand, flow dependences may have a severe impact on the achievable speedup since a data dependence violation will result in a roll-back. If a thread has computed a value before a more speculative thread reads it, the most recent value will be forwarded to the more speculative thread; but, if the value is computed after the more speculative thread performs the read, a flow dependence violation occurs. After a violation, the more speculative (violating) thread, will roll back execution in order to maintain a correct sequential execution. The model used here is capable of perfect roll-backs, which means that the thread causing the violation will be able to restart execution exactly at the load instruction causing the violation. Threads started by the violating thread after the erroneous instruction are squashed.

Flow dependences take two forms: flow dependences through memory and return values. To separate out the relative frequency of each category, I experiment with six alternatives:

- Memory accesses have either (1) perfect value prediction or (2) none at all. Perfect value prediction means that the prediction is always correct, and consequently there are never any memory-bound dependence violations.

- Value prediction for return values comes in three flavors: (1) Perfect return value prediction (RVP) is once again always correct; (2) stride RVP is supported by a table storing a last value and a stride value for each procedure; the table is of unbounded size. Predictions are updated in execution order, which is not necessarily in the same order as in the sequential execution. Additionally, it might happen that a finished thread updates the value predictor and then gets squashed, resulting in predictor pollution; one could say that the value predictor is speculatively updated. (3) The third option is no return value prediction.

With this model, questions related to the relative importance of memory versus return value flow dependence violations and how they relate to the programming style can be answered. One hypothesis would be that object-oriented programs tend to better encapsulate memory-bound flow dependences whereas dependences caused by return values become more critical. In addition, the model makes possible to pinpoint whether it would make sense to focus future research on more sophisticated value prediction schemes for thread-level speculation.

Model 3: Impact of limited processing resources

While the scale of CMPs will increase with increased integration, it may not make sense to devote too much resources for exploiting thread-level parallelism, for example when trading off number of processors versus issue-width. In the third model, I study to what extent the number of processors limit the speedup.

When the number of available threads exceed the number of processors, priority will be given to threads based on sequential execution order. While the number of processors are limited in this model, the number of thread contexts is still unlimited, and it is assumed threads can be preempted.

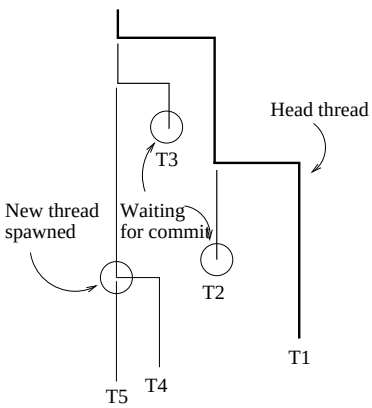


Figure 3.2: *Example of the advantage of preemption.*

Allowing preemption means a new thread can be spawned even if there are no free processors; one of the running more speculative threads are switched out and will remain dormant until there is a free processor available again. The rationale for prioritizing less speculative threads is that less speculative threads should be less likely to suffer from dependence violations, since they are closer to the non-speculative thread. In addition, they will be committed sooner and clear up space for new threads.

If preemption is impossible, a new thread can not be created when all processors are in use, unless a more speculative thread occupying one of the processors is squashed, wasting the work it has already done.

An even more serious consequence would be load-imbalance. The example in Figure 3.2 shows how two threads, T2 and T3, have completed their execution but must wait for the head thread, T1, to complete before they can commit. If the TLS machine has four processors running threads T1-T4 and the finished threads can not be switched out, thread T5 could not have been spawned. With this capability, one of

the dormant threads T2 and T3 can be switched out from its processor. The processor can then execute T5 instead of idling until T1 has finished.

In this model, the aim is to establish an upper-bound on the available parallelism given a certain number of processors. Therefore, preemption is allowed.

Model 4: Impact of limited thread contexts

Since a TLS machine must store speculative values from all threads that have not yet committed, the need to handle speculative state is perhaps the main reason why most proposed TLS architectures allow only a low number of threads in the system. Implementing efficient speculation mechanisms with a larger number of threads than processors is more tricky. In addition to the storage problem, threads that are not running must take part in dependence checking, value forwarding, and might need to roll back. The non-committed threads that exist must be visible to the speculation system even when they are not running on a processor. There must be support for at least one thread context per processor, i.e. where the running thread stores its speculative values. As mentioned, it is important to be able too keep non-committed threads in the system to alleviate load imbalance problems, and also if it is found desirable to allow preemption of running threads.

While many proposed TLS machines can only handle one speculative version per processor, and must wait until that thread has been committed before the processor is reused, there are more flexible designs. Hydra [HWO98] stores state in dedicated buffers and could be configured to support more threads than processors by adding more buffers than threads. One design from Steffan and Mowry [SCM97] makes it possible for each processor to handle multiple threads with a specific hardware structure (speculative context) for each thread. However, none these will scale to handle a large number of live threads due to the need for substantial additional hardware structures for each thread. A design by Renau [RSC+05], however, is able to manage a large number of simultaneous threads, this design was discussed in Section 2.2.3.

This model will be used to investigate how many thread contexts the architecture needs to support in order to exploit the available module-level parallelism. I will continue to assume that thread preemption is possible and run simulations with support for both infinite and a limited maximum number of thread contexts. When a call instruction is encountered and a free speculative context is available, a new thread will be spawned. When the thread limit has been reached, and a new call is encountered, the policy used is to start the new thread and squash the currently most speculative thread.

Model 5: Impact of thread-management overhead

In the preceding models, I have assumed that threads can be spawned, committed, and rolled back in zero time. On recently proposed CMPs such as Hydra, the overheads imposed by these operations are not negligible. To what extent the overheads have a significant impact on the speedup obtained is strongly connected to two application parameters: the number of flow dependence violations and the module granularities.

The goal with this model is to factor in these overheads to identify what mechanisms would have to be further researched to come up with machine models better adapted to module-level parallelism. Fixed-length overheads are added when spawning, restarting, or committing a thread, as well as when performing a context switch.

Model 6: Roll-back policy

One reason why roll-backs are especially harmful is because in addition to the overhead of the roll-back handler code, useful work is being thrown away. How much work is thrown away depends on how accurately the roll-back mechanism can squash work affected by the violation. An optimal mechanism would only squash and re-execute instructions that depend on the erroneous value. However, this would entail tracking the effects of the erroneous value through the violating thread and its child threads, which is not practically feasible for most architectures.

The mechanism which is most straight-forward to implement, and commonly suggested for TLS machines, is to squash the entire thread containing the violating instruction, as well as any thread spawned from this thread. I call this method *thread roll-back*, since the entire violating thread is squashed.

The method used in the previous models is to re-execute everything after the violation, but save work performed by the thread prior to the violation. This method would require checkpoints before every exposed load and used return value, so that the roll-back mechanism could restore the correct state at any such event. The difference between thread roll-back and this method, which I call *perfect roll-back*, is visualized in Figure 3.3. In the figure, the dotted lines represent the part of the threads which are squashed after the indicated dependence violation. At each checkpoint, the register contents must be backed up and the speculative memory state after the checkpoint must be separated from the state prior to the checkpoint. A checkpoint mechanism has been described by Olukotun et al. [OHW99].

Since checkpointing requires extra time and resources, it could be done on a select few loads instead of every single one. Olukotun et al. save checkpoints before violation-prone loads. Another alternative could be to checkpoint before a load which occurs after a certain amount of cycles have passed since the previous checkpoint. This would at least decrease the average roll-back distance.

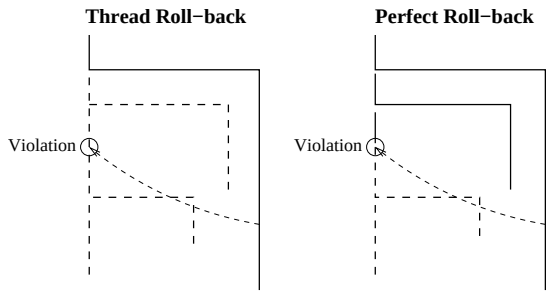


Figure 3.3: *The violation causes the dotted part of the thread to be squashed.*

3.2 Simulation Methodology

A simulation framework has been developed in order to investigate the described architectural models. In this section, the framework is described, the parameters for the baseline model are listed, and the benchmarks used for the experiments are presented.

3.3 Simulation Tools

The simulation tools implement all architectural models described in the previous section. Figure 3.4 summarizes the simulation framework. First, applications are compiled with GCC. The compiler inserts annotations which are used in the next simulation phase; running the applications sequentially. The applications are executed sequentially on Simics, a system-level instruction set simulator, Simics [MLM+98]. The annotations work as call-backs to Simics, and are used to save needed information about the execution to a trace file. The trace file is used by my TLS simulation tool.

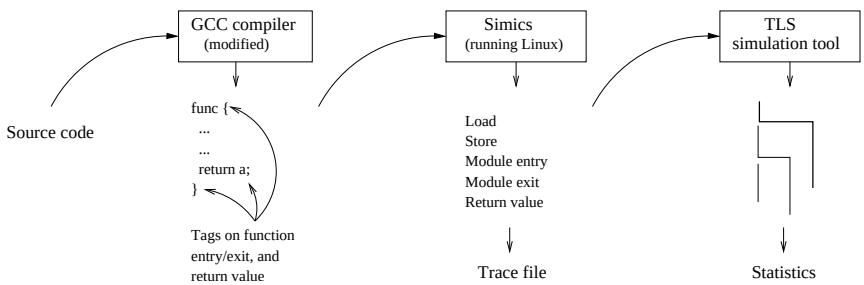


Figure 3.4: *The simulation toolchain.*

Simics makes it possible to run applications and OS in a simulated environment, and to capture memory accesses and register contents without introducing any overhead in the application. Simics can call a user defined function when encountering a memory access or a special instruction, i.e. the annotations inserted by the compiler. This feature was used to capture a trace containing memory accesses, module calls and returns, as well as return value and the first occurrence of return value use. Each instruction is tagged with a time-stamp showing in which cycle it was executed.

The TLS simulation tool runs a program much as it would run on a real machine with speculation support; that is, threads are executed in parallel with run-time dependence checking. If a dependence violation is detected, the violating thread is rolled back and subsequent threads are squashed.

As opposed to a real machine, only instructions of importance for the simulation are supported; i.e. the instructions captured in the trace. A virtual timer, which keeps track of the number of instructions executed between such events, is associated with each thread. With the virtual timer and the time-stamps in the trace, the simulator can fake execution of the correct number of instructions between each of the supported instructions, even though the instructions are not actually recorded in the trace.

The simulated processor is a single-issue in-order SPARC v8. The memory system is assumed to be perfect, loads and stores are always available for use in the next clock cycle. This means that the simulated system always completes one instruction each cycle. Realistic processor core and memory hierarchy models would affect the run-time of each module and therefore the end result of TLS execution. The impact of these parameters is covered from Chapters 7 to 9.

The programs were compiled with the GNU Compiler Collection (GCC) 2.95.2 with full optimizations. In a Java Virtual Machine (JVM) environment, the execution of a Java program includes class loading and verification, Just-In-Time (JIT) compilation and/or interpretation, and garbage collection. In my measurements, the Java programs are run without a JVM. Instead, they are compiled to native executables, which means neither class loading and verification, nor interpretation or JIT-compilation occurs. Furthermore, garbage collection has been disabled. Since my intention is to find the parallelism inherent in the applications, and not to evaluate the Java run-time system, this method makes sure the measurements only contain code execution. In addition, it gives a fair comparison between Java (Object-Oriented) and C (Imperative) codes since GCC can be used to compile all of the benchmarks. It should be noted, however, that the Java compiler in this version of GCC is under development and the optimizations are likely not as advanced as for the C compiler. Therefore, the instruction counts for the Java applications might be somewhat higher than what could be achieved with a production-quality compiler.

Only modules in the actual application are marked by the compiler. This means the tools do not speculate on library (or class library for Java) calls. Such functions

are run inside the caller thread. Another noteworthy detail is that exceptions and I/O operations would inhibit the ability to run threads speculatively in a real TLS machine, and therefore speculation must be suspended during these events. However, such events are rare in the benchmarks used. Finally, artificial dependences through the stack between modules executing in parallel have been filtered out.

3.3.1 Baseline TLS Machine

Since the aim is to find the limits of module-level parallelism, the baseline TLS chip multiprocessor system implements the most desirable qualities for TLS systems described in Section 2.2. The major features of the baseline system are summarized in Table 3.1. Section 2.2 also describes possible solutions for implementing such a system. However, details of an implementation are left out for now. Implementation issues are discussed in Chapter 7.

Table 3.1: *Baseline speculative chip multiprocessor.*

Processor cores	*n*-way chip multiprocessor with single-issue in-order cores and 1-cycle memory accesses (i.e. CPI=1).
Overhead	Thread-starts, roll-backs, context switches, and commits are modeled with fixed-length overhead. The size of each overhead is configurable.
Value prediction	Stride value prediction is used for module return values. Perfect return and memory value prediction is used in some experiments in order to establish an upper bound on speedup. The stride predictor uses an infinite prediction table.
Thread scheduling	Threads can be preempted. On an *N*-way machine the *N* least speculative threads are always scheduled to run. There is no limit on the number of threads active and waiting to run, except in the thread context experiment.
Data dependences	Violations due to anti- and output dependences are avoided with renaming. Flow dependence violations are avoided with forwarding whenever possible. Unavoidable violations cause a roll-back and restart of the dependent threads. Perfect roll-back is used except in the thread roll-back experiment. Buffer space for speculative state is unlimited.

3.3.2 Benchmarks

Ten benchmarks have been selected, four written in C (imperative) and six written in Java (object-oriented). The C benchmarks are from the well-known SPEC CPU 95 integer benchmark suite (CINT95) and have been used in earlier TLS limit studies [OHL99, MG00]. From the eight CINT95 benchmarks, four programs that based on the earlier studies seem to represent typical behavior are chosen.

The **Gcc** application is the GNU C compiler 2.5.2 compiling a single source file, and **Compress** is the Unix compress utility. **Go** is a simulation of the board game go, and **M88ksim** simulates an m88k processor, running the Dhrystone benchmark on top of the simulated processor.

Three of the Java benchmarks are from SPEC JVM98. **Compress** is a Java version of the compress utility, **Db** simulates a simple database, and **Jess** is a simple expert system. Unfortunately, the rest of the benchmarks in the suite did not include source code which is needed for the compilation/annotation phase in the simulation framework. Instead, two benchmarks from jBYTEmark (also used in [CO98]) and the constraint solver benchmark **Deltablue** from Sun Labs are included. The jBYTEmark applications are **Idea**, which does encryption and decryption with the IDEA block cipher algorithm, and **Neuralnet**, a back-propagation neural network simulation.

All these applications are general integer applications which have been found difficult to parallelize with conventional methods. Integer applications often suffer from poor parallelism coverage, lack of do-all loops[1], and complex data- and control flow.

In order to keep simulation times down, the size of the input sets had to be small. Therefore, all data sets have been pruned down so that the running time of the application is only a few million cycles. While the data sets are small, there are still plenty of module calls to speculate on. However, it should be noted that using larger input sets typical for real-world use of these applications could affect the result on some of the benchmarks.

Table 3.2 shows some statistics for each application, namely dynamic instruction and module counts, as well as average module size and static module count. However, it should be noted that the average module size can be a bit misleading; module sizes vary greatly. Section 3.4.5 will show that a majority of modules are less than 100 instructions in all but two of the programs.

[1] Do-all loops are loops without loop-carried dependences, which means they can easily be parallelized even without TLS support.

Table 3.2: *The benchmark applications.*

Name	*#Instructions (dynamic)*	*#Modules (dynamic)*	*Avg. instr./mod. (dynamic)*	*#Modules (static)*
C Applications				
compress	1.4M	21k	67	8
gcc	13M	54.5k	237	525
go	1.4M	1.1k	1190	105
m88ksim	2.2M	0.5k	4767	34
Java Applications				
compress (Java)	2.7M	31.5k	84	47
db	13M	4.9k	2644	52
deltablue	2.6M	12.5k	208	76
idea	35.7M	12k	2966	16
jess	16.3M	25.8k	633	484
neuralnet	4.2M	2.6k	1626	26

3.4 Experimental Results

I begin with studying the upper bound on the module-level parallelism in Section 3.4.1 followed by the impact of data dependences in Section 3.4.2, limited number of processors in Section 3.4.3, and limited contexts in Section 3.4.4. The impact of thread-management overhead is studied in Section 3.4.5, and finally the roll-back policy is investigated in Section 3.4.6.

3.4.1 Limits on the Inherent Parallelism

Figure 3.5 shows the speedup for the benchmark applications with perfect (i.e. always correct) value prediction both for return values and all memory loads. The geometric means for both groups (C and Java) of applications are also included. The speedup under ideal machine conditions is only limited by the module-level parallelism inherent in the program structure as constrained by the control dependences, i.e. how often and when modules are called. Figure 3.5 therefore serves as a fundamental limit for module-level parallelism, given the simplistic execution model where a speculative thread is spawned whenever a module call is encountered. The only way to increase the module-level parallelism would be to hoist the thread spawn so that the treads are started before execution has reached the point of the call, a more complex solution. To some extent, compiler transformations could also rearrange the calls in a more advantageous way, i.e. to increase the overlap of module execution.

Mean speedup without the impact of dependences is close to 3.5 for both groups (excluding Neuralnet), though somewhat higher for the Java applications. This means

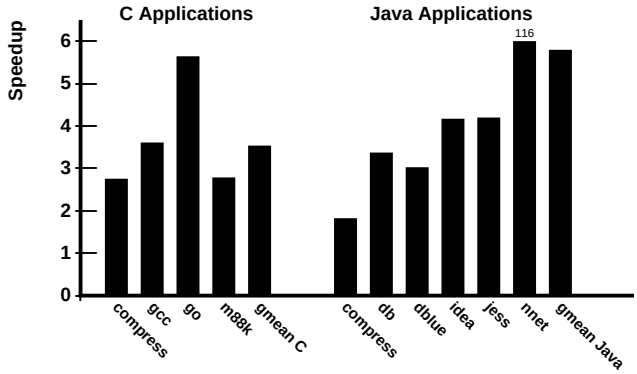


Figure 3.5: *Speedup on the ideal machine with perfect memory and return value prediction.*

that the module calls are not arranged in such a way that there will be a large overlap of modules even if all calls are parallelized. A contributing reason to why there is no large additive effect is that the majority of modules are small. However, if some of the parallelism can be extracted with reasonable effort, it might still be a useful proposition given the simplicity by which this parallelism can be extracted from existing programs.

The reason for the high speedup (116) in Neuralnet is that a number of modules are called repeatedly inside a tight main loop, encompassing the entire program except for a short initialization phase. Thus, the main loop uncovers large amounts of module-level parallelism.

There are no significant differences between the Java and C applications with respect to potential module-level parallelism, the available parallelism is about the same for both programming styles.

3.4.2 Impact of Data Dependences

The previous model predicted speedup under the assumption that value prediction on return as well as memory values is perfect. Perfect value prediction is of course not possible to attain, so the first question on my trek towards a realistic machine model is: how would value predictors with reasonable implementation complexity affect speedup?

Figure 3.6 shows how memory load value prediction (MVP) affects performance. For each application the left bar, labeled (P), shows the speedup with perfect MVP, while the right bar, labeled (N), indicates speedup with no MVP. The difference in height thus shows the potential of memory load value prediction.

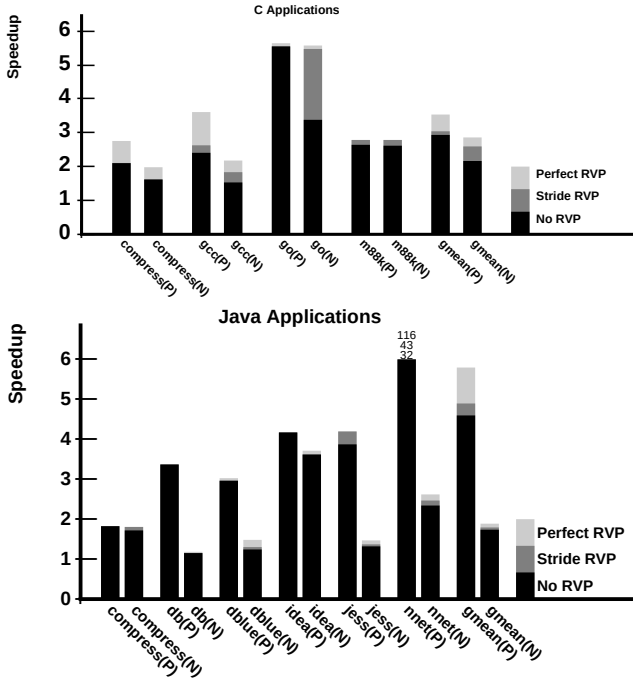


Figure 3.6: *Value prediction: the left bar (P) for each application has perfect memory value prediction, the right bar (N) has no memory value prediction.*

The lack of memory value prediction has a substantial impact on some of the benchmarks. For instance, most of the massive potential in the Neuralnet benchmark disappears. Neuralnet contains numerous shared data structures that are continuously updated in each iteration of a main loop; therefore, this main loop is not possible to parallelize. The remaining parallelism comes from partial overlap of modules within a loop iteration. The key methods in Neuralnet do contain a good amount of loop-level parallelism, which cannot be exploited with the module-level approach. Chen and Olukotun [CO98] have extracted parallelism from this application by modifying the code, converting loop-level parallelism to module-level parallelism.

The shaded vertical sections on each bar in Figure 3.6 show the impact of return value prediction (RVP). Three policies are presented: no RVP, stride RVP, and perfect RVP. For the no RVP policy, modules are still run speculatively, but a roll-back always occurs to the point where the return value is used. The gap between no RVP and perfect RVP will reveal the potential benefits of return value prediction. I also included a known and computationally simple value predictor as an indication of the

predictability of return values; the stride predictor, which predicts the next value as the last value plus the difference between the two last values. Another obvious candidate would be a last-value predictor, however, both catch the most obvious case of a function that almost always returns the same value (the stride would be zero).

Return value prediction seems to make sense in some of the benchmarks, but surprisingly, in many of the programs most of the parallelism can be exploited without RVP, since a large portion of the modules either do not produce a return value at all (void modules), or produces a return value which is never used. If one would choose a scheme without RVP, a speculation system could catch and roll back modules in the cases where the return value is indeed used, but a better way would be to have the compiler mark all calls to void modules and those whose return value is not used, since this can be determined statically.

The simple stride value predictor has been observed to perform reasonably well in most applications, successfully predicting between 20% and 80% of return values in seven of the ten applications. Some applications, notably Idea and Neuralnet, did show a very large percentage of mispredictions. It turned out to be because of heavy use of a random number function during initialization (which is a small part of the total execution time). It would be useful to be able to selectively disable speculation for such cases, where obviously no value predictor can be expected to perform well.

The execution time in Idea is concentrated to one module which handles encryption/decryption. Most of the speedup for this program originates from overlap of two iterations of encryption and decryption (four calls to this module). Although it is written in Java, it has the structure of an imperative program, which is not surprising considering the fact that it is converted from C. Neuralnet and Java Compress are also originally C programs.

My initial belief was that the object-oriented (Java) programs would exhibit more parallelism than the imperative (C) at this point for two reasons: the object-oriented programming style encourages more frequent use of module calls, and the use of data encapsulation would result in fewer memory dependences. However, the results do not indicate any such difference. There seems to be a small difference when it comes to return values, the speedup of the C programs are slightly more affected by roll-backs due to return value mispredictions.

In summary, two important lessons can be learned from this experiment: a simple return value predictor will suffice in most cases, and a good memory value predictor would be very useful. In the rest of this chapter no value prediction on memory loads, and stride value prediction for return values will be assumed. This should represent a design choice of reasonable complexity.

3.4.3 Impact of Limited Processing Resources

In Figure 3.7 speedups for the applications running on a machine with limited processing resources can be seen. The model is still ideal in the sense that I assume the processors in an *n*-way machine can always be utilized executing the *n* least speculative threads. A more speculative thread will be preempted, without penalty, if a new less speculative thread arrives; execution will be resumed, however, where it was preempted the next time the thread can be rescheduled on a processor.

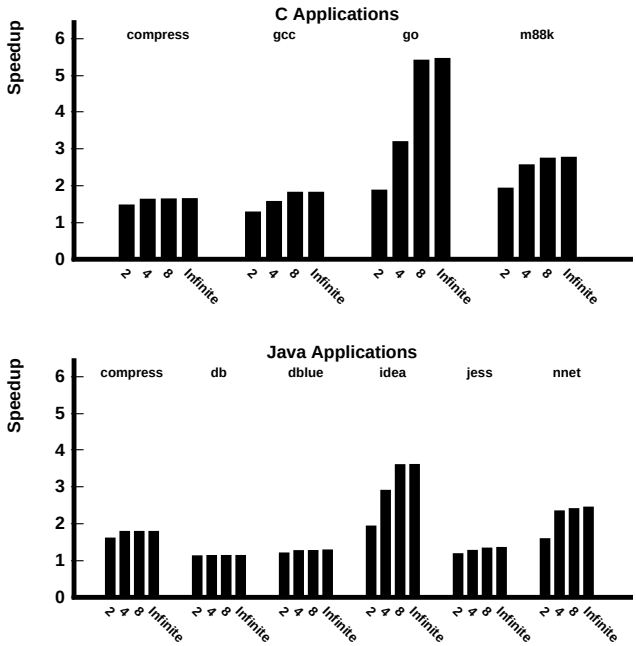


Figure 3.7: *Speedup with 2, 4, 8 or an infinite number of processors.*

With this model, it can be seen that virtually all potential speedup can be utilized with only eight processor cores. In fact, many of the benchmarks do not benefit significantly from more than four processors. This is good news, since it shows that parallelism is in general not concentrated to a limited part of the execution; rather, a limited number of processors are busy working most of the time.

This data suggests that all the module-level parallelism available in C and Java programs could potentially be exploited using chip multiprocessors with relatively few processor cores. Again, there is not any big difference across C and Java programs.

3.4.4 Impact of Limited Thread Contexts

Speedups when limiting the number of thread contexts to 8, 16, or 256 are shown in the first three bars for each benchmark in Figure 3.8. The results are compared to the speedup with an unlimited number of contexts, which is shown in the fourth bar. In this figure, I do not separate the C and Java benchmarks, and I have omitted the Java version of compress as it has shown to behave much like the C version. An 8-way machine model is used, since this turned out to be sufficient to exploit most of the available parallelism.

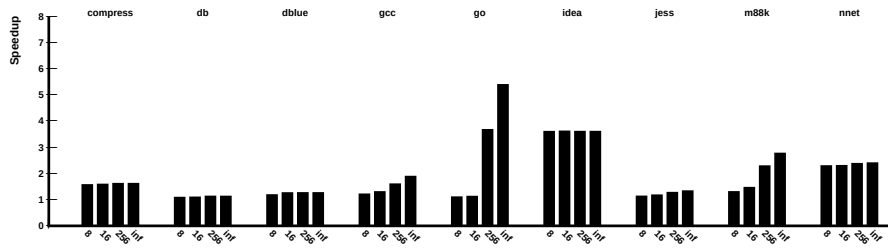


Figure 3.8: *Performance with limited thread contexts on an 8-way machine.*

It is clear that for some applications, in this case Gcc, Go, and M88ksim, depend on an adequate number of thread contexts to reach their full potential speedup. These applications show no or very little speedup with only 8 or 16 thread contexts for the 8 cores. With 256 contexts, most of the potential speedup can be exploited, but even more are necessary to reach the full potential.

The results show that is is indeed important to support more thread contexts than the number of processors; this to reduce the effect of load imbalance and make preemption of less speculative threads possible.

3.4.5 Impact of Thread Management Overhead

Figure 3.9 shows speedup with overhead for speculation support. I have included four types of overhead: starting a new speculative thread, performing a roll-back on misspeculation, committing speculative state when a thread has successfully finished, and context switch overhead. A thread that has been squashed as part of a roll-back will incur a new thread-start overhead when it is called again.

In the figure, all types of overhead are set to the same size; I ran simulations for 10, 100, or 1000 cycles. For the sake of comparison, the speedup values for the no-overhead machine are repeated. The numbers are for an 8-way machine.

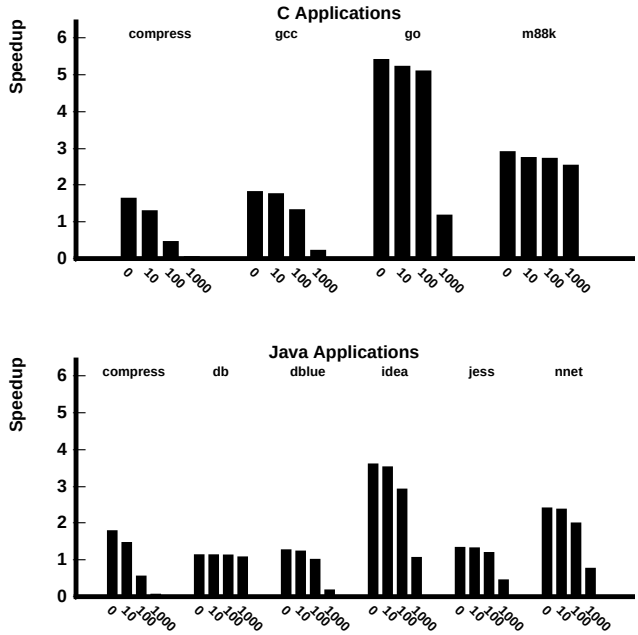
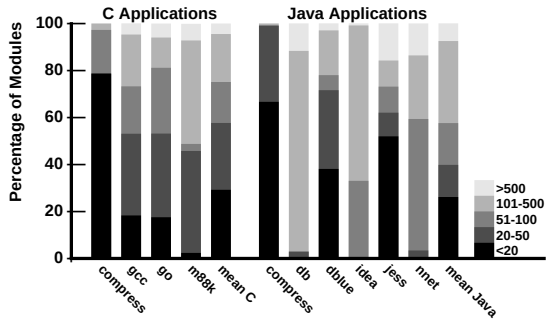


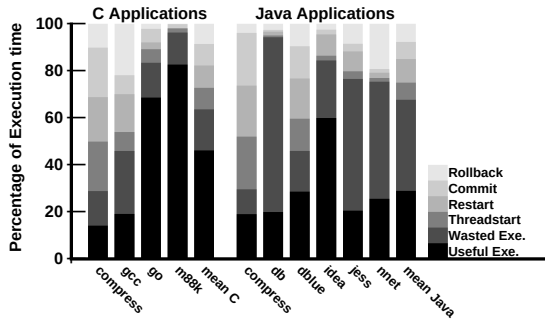
Figure 3.9: *Speedup with thread-management overheads of 0, 10, 100 or 1000 cycles on an 8-way machine.*

The 100-cycle overhead simulations are interesting since they approximately correspond to the overheads reported for module speculation support in the Hydra CMP [HWO98]. At 100-cycle overheads, there is already a severe impact on the speedup for several applications, even a slowdown for both Compress programs. When the overhead is increased to 1000 cycles, the Compress programs are more than ten times slower than their sequential execution.

A reason for this is module granularities. Figure 3.10(a) reveals that for both C and Java Compress, the majority of the modules are shorter than 20 instructions, and almost all are under 100 instructions. This means that thread-management overheads will dominate execution time, since each module will, at least, give rise to a thread-start overhead when called, and a commit when it reaches return. On the other hand, some of the modules are very large, which explains why the average sizes presented in Table 3.2 are several thousand instructions for some of the programs. Note that with a single-issue, perfect memory machine, there is a one-to-one correspondence between the number of cycles and instructions.



(a) *Percentage of modules with dynamic size in ranges from <20 to >500 instructions.*



(b) *Part of execution time that is useful, wasted because of roll-backs, and used for thread-management.*

Figure 3.10: *Statistics for an 8-way TLS machine with 100-cycle overheads.*

An observation is that one of the side effects of increasing the number of processors is that the number of dependence violations will also increase. Therefore, with high thread overheads, the benefits of adding more cores to the CMP will be smaller than indicated in Figure 3.7, in some cases there can even be a negative effect of adding more cores.

In Figure 3.10(b) one can see how the execution time is used. The execution time for a program in this figure is the total used time on all processors added together. The simulations are run with 100-cycle thread-management overheads on eight processors.

Useful execution is the part of the execution that was successful and committed; it is the part that corresponds to the sequential execution. Wasted time is the execution time that was thrown away because of a roll-back or when the thread was squashed. Restart is the effect of additional thread-start overhead for a thread that was squashed

and must be started again. The remaining three categories show thread-start, roll-back, and commit overhead.

This figure points out one of the serious disadvantages of speculative execution. Only 20% on average for Java programs, or 40% for C programs, of the processing time is useful execution. This is a disadvantage in a multitasking environment where other processes might make better use of the resources. It is also a problem from an energy-efficiency perspective. Wasted execution makes up a major part of the total processing time for most benchmarks. A conclusion would be that methods for minimizing the number of misspeculations, and thus wasted execution, is very important in a TLS system even if it is not necessary from the point of view of performance for a single-application.

It is also clear from this experiment that keeping overheads small is of utmost importance for module-level speculation support. With well-tuned hardware support, 100-cycle overheads is somewhat pessimistic. From Chapter 7 on, a more detailed model of overhead with faster thread-management operations will be used.

3.4.6 Significance of Roll-Back Policy

Figure 3.11 shows the performance difference between thread roll-back and perfect roll-back. The grey bars show speedup with perfect roll-back, and the black bars are for thread roll-back. An 8-way machine with overheads of 100 cycles is used.

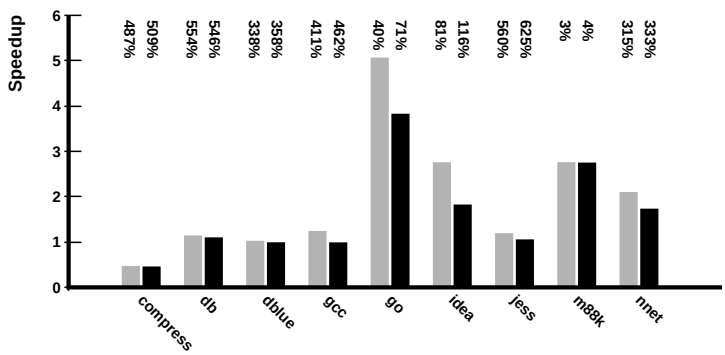


Figure 3.11: *Performance of perfect (grey) vs. thread (black) roll-back, 8-way machine with 100-cycle overheads.*

Some of the programs, notably Go and Idea, suffer from the less precise thread roll-back; in total, five of the applications have notably lower speedup. As opposed to the results reported by Olukotun et al. [OHW99], which focused on loop-level

parallelism, the coarser thread roll-back can have a detrimental impact on the speedup of module-level parallelism for my benchmark programs.

In conclusion, if there is a way to identify where checkpoints should be saved, a checkpoint mechanism would indeed be useful for module-level parallelism.

3.5 Related Work

Value prediction enables speculation beyond the data flow limit. It was introduced by Lipasti et al. [LWS96] as a way to hide memory load latency by allowing data dependent instructions to execute in parallel. The predictability of data values was investigated by Sazeides et al. [SS97]. Others have followed up with a number of innovative prediction schemes.

This chapter focuses on the opportunities and limitations of speculative module-level parallelism – a straight-forward method to extract thread-level parallelism out of existing software. Several prior papers have had similar goals. A limit study of the inherent loop-level as well as module-level parallelism in CINT95 applications has been published by Oplinger et al. [OHL99]. While disregarding architectural limitations in terms of thread management overheads, they found that there is ample module-level parallelism in the CINT95 that can be exploited by multiprocessor or multithreaded processor cores of typically less than eight processors. In comparison with my study, they did not address how important memory-level dependences are and did not look at Java applications. Moreover, they did not study how much typical overheads in CMP architecture models would affect the achievable speedup.

In contrast, Chen and Olukotun [CO98] focus on Java programs. Their study is mostly aimed at the speedup obtained on the Hydra CMP proposal and does neither address the impact of various value prediction schemes nor how scalable the parallelism is. While they note that thread management overhead may have a severe impact on the speedup, they did not analyze how it relates to the size of the modules.

In a follow-up study by the Hydra team, based on CINT95 programs [OHW99], they observe that thread management overheads can be detrimental to the speedup obtained because of the penalties associated with misspeculations. As a remedy, they propose and evaluate schemes that select modules to speculate on depending on their likelihood to succeed.

Value prediction as a way to reduce dependence violations in thread-level data dependence speculation architectures has been investigated by Marcuello et al. [MTG99, MG00] in the context of their Clustered Speculative Multithreaded processor. They speculate on live input values to threads (values used but not defined within the thread) at thread start time. Some works mentioned earlier also used value prediction for module return values [CO98, HWO98] and memory loads [OHL99]. Others who have used value prediction in conjunction with coarse-grained speculative

architectures include [RJSS97, AD98, CW99a]. Later, Hu et al. [HBJ02] investigated value prediction for return values. However, to the best of my knowledge, the study in this paper was the first to address the limits on value prediction and pin-point whether there is room for improvements.

3.6 Conclusions

The goal of this study has been to understand the impact of the programming style – imperative versus object-oriented – on the inherent module-level speculative parallelism as well as how architectural deficiencies in proposed chip-multiprocessor architectures affect the achievable speedup.

One would expect that object-oriented programs would make more heavy use of modules and would encapsulate many of the data dependences with a potential to expose more module-level parallelism. Contrary to this intuition, I found that there is not any significant difference between the inherent module-level parallelism in the C versus the Java programs that I studied. In both cases, the speedup limit was about 3.5 on average. In addition, applications from both programming styles were sensitive to memory-level data dependences, which suggests that progress in memory value prediction schemes are important to approach the maximum speedup. As for return-value prediction schemes, simple ones based on last- or stride-value fare pretty well across all applications.

When considering the impact of architectural constraints, I found that all of the inherent parallelism could be exploited by small multithreaded or multiprocessor machines with eight processors, provided more thread contexts and preemption is supported However, a key bottleneck is the overheads imposed by thread management, including the time to start (or restart), commit, or roll back threads after data dependence violations. Given the fairly small module sizes, speedup is severely affected when the overheads exceed a hundred cycles. This means efficient thread management mechanisms are vital. Obviously, using module-level parallelism in more loosely coupled architectures is not an option. Finally, the roll-back policy used does have an impact; if it is possible to implement a check-point mechanism efficiently, it would be useful for module-level speculation.

In this study, I did not try to enforce a certain thread granularity, instead all modules were parallelized regardless of size. On realistic architectures, with various overheads, granularity is important. Methods to selectively apply module-level speculation are needed. My reason for using modules as the only source of parallelism was that they are mostly control independent and easy to identify. Since the amount of module-level parallelism is limited, additional sources of parallelism are needed in order to achieve large performance gains with thread-level speculation techniques.

This chapter is a revised version of the previously published paper "Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction" [WS03].

4
Run-Length Prediction

In Chapter 3 I found that programs from integer benchmarks, CINT95 and SPEC JVM98 to be specific, have a speedup limit ranging from two to six with module-level parallelism on an eight-processor CMP. However, achieving this speedup is mainly limited by the overhead associated with thread management and misspeculations.

Thread creation and termination, roll-backs, and context switches are all associated with some overhead. If the overhead is significant in comparison with the module execution time – or run-length – the contribution to the overall speedup is small. Hammond et al. [HWO98] (Hydra) found threads of size 300-3000 instructions suitable if overheads are in the range 10-100 cycles. Consequently, using the run-length of a module as a key criterion for selecting which modules to speculate on appears to be a promising way to reduce the impact of thread management overhead. The potential of this technique is explored in this chapter.

I first investigate how much speedup can be gained by only speculating on modules with a run-length greater than a certain threshold. Based on nine Java and CINT95 applications, it is possible to eliminate most of the impact of overhead in the range of 100-500 cycles on speedup by only speculating on modules whose run-length is above a certain threshold, typically around 500, assuming perfect a priori knowledge of the run-length.

I then introduce the design of a module run-length predictor that, based on the previous run-length of the module, will predict if future invocations of the module

will exceed the threshold or not. This predictor is shown to behave very close to the off-line omniscient predictor with a prediction accuracy between 83% and 99%. I demonstrate that such a predictor can wipe out almost all of the impact of thread-management overhead on the overall speedup of the applications on an 8-way chip-multiprocessor with support for TLS. As opposed to related off-line techniques such as compiler inlining[1], my method can be used for run-time speculative parallelization of sequential binaries.

Finally, I apply the run-length predictor to machines with a limited number of thread contexts. Two benchmarks benefit significantly from the run-length predictor with limited number of contexts, this since the number of threads spawned is decreased.

In Section 4.1 module run-length thresholds and their impact on overhead penalty are investigated. Section 4.2 introduces the run-length predictor, and in Section 4.2.2 its performance is compared to that of a perfect predictor. Section 4.3 discusses the problem with thread contexts, and finally some conclusions are drawn in Section 4.5.

4.1 Potential of Run-Length Thresholds

In order to demonstrate the impact of thread management overheads on the potential speedup of speculative module-level parallelism, simulations with thread-start, roll-back, and context switch overheads were run. In Hydra [HWO98], speculation events are handled by a speculation coprocessor where control routines of typically 50-100 instructions are executed for each event. While these overheads are useful as reference points, it is unclear how many cycles of overhead future TLS machine implementations will have. Therefore, overheads ranging between zero and 500 cycles per event are used in this chapter in order to study the sensitivity of the overhead impact on speculative module-level parallelism.

In Figure 4.1 the speedups of the nine applications for different overheads are shown. The upper graph shows simulations with the perfect value prediction model, and the lower graph with realistic value prediction. This figure is similar to Figure 3.9, but instead of 1000-cycle overheads, results for more moderate 200-, and 500-cycle overheads are shown. In addition, 10-cycle overheads are not used, since the aim is to investigate if run-length prediction can be used to mitigate the effect of relatively large overheads.

For overheads of 100 cycles, the speedup is already severely hampered, especially under the realistic model where roll-backs and thread restarts kick in. Moreover, with a 500-cycle overhead, speedup is more than halved for most applications. M88ksim is less affected by roll-backs, and thus experiences less events causing overhead. Com-

[1] Note that the applications used in the experiments are compiled with inlining activated as well.

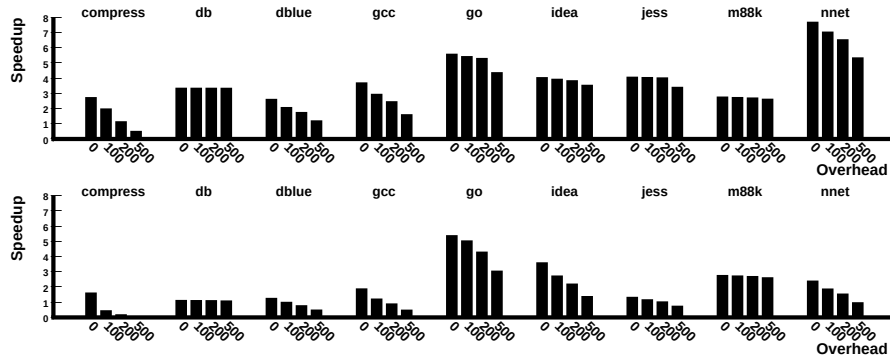


Figure 4.1: *Speedup with thread-management overhead 0-500. The graphs show results with perfect (upper) and realistic (lower) value prediction models.*

press, on the other hand, which largely consists of very small modules, already suffers from a slowdown at 100-cycle overheads.

4.1.1 Basic Idea

In order to better amortize the overhead costs over the useful execution, one want to avoid spawning off new threads which do not contribute to the speedup or worse, tie up machine resources with little gain. I do this by applying a threshold on the module run-length. If the run-length exceeds the threshold, a new thread is created for the module continuation. If not, the overhead is expected to negate any positive effect of the gain in parallelism, so the code is run sequentially.

Module run-length is defined as the time between the call and return of a module. As shown in Figure 4.2, this time will include the run-time for child modules run sequentially, but exclude run-time for child modules when new threads are created. Overhead is not included in the run-length, only useful execution. The module run-length is related to how much useful parallel overlap the execution a new thread can yield.

4.1.2 Simulation Methodology

The simulation setup and benchmarks are the ones described in Section 3.2. The simulator has been extended to assess the potential of run-length thresholds as well as include the prediction techniques described in this chapter. Some parameters used in the experiments are listed in Table 4.1.

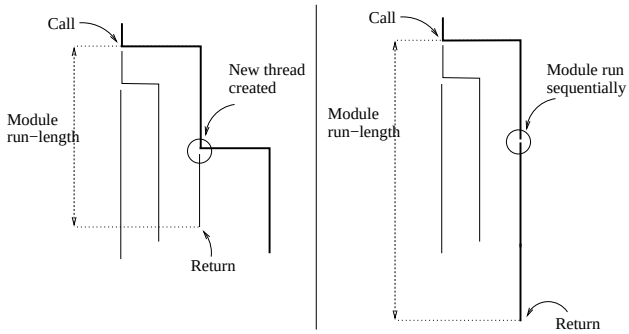


Figure 4.2: *Module run-length calculation.*

Table 4.1: *Baseline machine parameters - run-length prediction.*

Feature	*Baseline parameters*
Processors	8
Overhead	100, 200, or 500 cycles for thread-start, roll-back, commit, and context switch.
Rollback policy	Perfect roll-back.
Value prediction	Stride return value prediction. No memory value prediction.
Run-length threshold	100, 500, 1k, 5k, or 10k cycles

4.1.3 Experimental Results

Since trace-driven simulation is used, the dynamic run-lengths can be precomputed from the execution traces in order to assess the potential of run-length thresholds. This a priori knowledge is used when applying different thresholds in this section. This is, however, not possible in a real-world system.

Figure 4.3 shows speedups for the benchmark applications with thresholds between 0 and 10000 cycles. Full speedup graphs are shown for Gcc, Go, and Neuralnet, and abridged versions (only three thresholds) for the remaining applications. Gcc and Neuralnet were chosen as good examples of the usefulness of module run-length thresholds, whereas Go is included to show some unusual behavior. In the full graphs, each line represents a different amount of overhead. The vertical axis shows speedup and the horizontal axis different thresholds. Note that the vertical scales are different for the applications.

In the bar graph, different overheads are depicted with shaded sections. The whole bar shows speedup for zero overhead. Then, progressively darker sections show speedup with 100, 200, and 500 (black section) cycle overheads respectively.

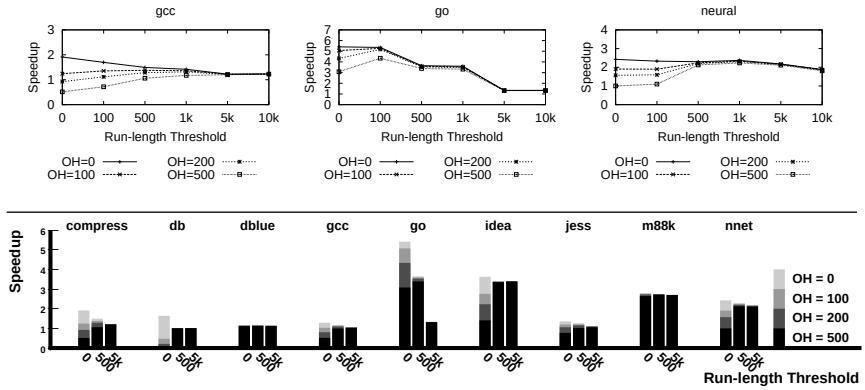


Figure 4.3: *Speedup with module run-length thresholds between 0 and 10000 cycles.*

For example, speedup for compress without a run-length threshold (or threshold=0) is: with zero overhead 1.64, for OH=100 it is 0.47, for OH=200 only 0.21, and for OH=500 it is 0.1.

A speedup improvement is achieved on all applications except Db and M88ksim. This is expected as Db and M88ksim have a larger portion of long modules and the impact of overhead is small. Jess and Deltablue show improvements, and a small positive speedup; without module run-length threshold they suffer a slowdown. Gcc suffers badly from misspeculations, which a run-length threshold does not solve. Compress hardly has any parallelism with a threshold of 100 or above. The run-length predictor effectively nullifies the overhead so that, at least, it runs sequentially with no overhead. Go has a lower best threshold than the other applications. The best result is achieved for a threshold of around 100; at 500 the speedup is down again due to a lack of parallelism.

Overall, for six out of nine applications, the speedup at an overhead of 200 is very close to the speedup without overhead when a good run-length threshold is used, and none of the programs suffer from slowdown.

4.2 Module Run-Length Prediction

In the previous section it was shown that creating new speculative threads only when the module run-length exceeds a threshold can help alleviate the impact of thread-management overheads. However, the decision to create a new thread needs to be done when the module is called, and we cannot know the run-length until it has completed execution. To overcome this problem, I make use of a technique common

in computer architecture: history-based prediction. It is reasonable to assume that there is a correlation between the run-length of one invocation of a module to the next.

4.2.1 Algorithm & Implementation

The predictor works like this:

- Each module in the application has its own predictor associated with it. The predictor uses a single bit which designates whether run-time was above or below the run-length threshold for the most recent completed execution of the module.
- The module run-length is measured every time the module is called. When it completes (reaches return), the measured run-length is compared to the threshold. If it exceeds the threshold, a '1' is stored in the predictor bit, otherwise, a '0' is stored.
- When execution reaches a module call, the prediction bit is checked. If the bit is '1', a new thread is created for the continuation, otherwise the module is run sequentially.
- All prediction bits are initialized to '1', so on the first invocation a new thread will always be created.
- Zero-latency is assumed for the prediction mechanism in the simulations.

Note that the run-length is measured regardless of whether a new thread is created or not; otherwise a module that has once been marked '0' would no longer be updated, and the prediction could never change. Since the result of prediction changes further down the call tree can propagate to parent modules, it is especially important that predictor changes can go both ways; it might take a few invocations before the predictor reaches steady state.

The possible advantages of measuring module sizes dynamically instead of doing static analysis is that the length may be hard or impossible to determine statically. In addition, a dynamic predictor can automatically adjust to hardware dependent parameters such as communication and memory latency. It is likely, however, that a combination could be useful. For instance, very small modules whose length can be determined statically could be removed from being considered for speculation, in order to minimize overhead from the run-length measurements.

In order to implement run-length prediction, methods for measuring the run-length as well as a structure for storing history bits and temporary cycle counts is

needed. Storage should be shared among the processors in the CMP in order to support preemption, and a shared predictor will faster become warm.

The storage could be implemented as a dedicated hardware structure, or in order to avoid extra hardware, in the memory hierarchy. As can be seen in Table 3.2, the number of unique modules is at most a few hundred. Therefore, the structure need not be very large, at least for these applications.

Most existing processors have performance counters, including a cycle-counter, which could be used for a software implementation of run-length prediction. Measuring the module run-length could be done by recording the cycle count at the module call, and comparing it with the count after completed execution. Care has to be taken, however, to exclude overhead and time when the module is not running, e.g. swapped out in favor of a higher-priority thread. Reading the performance counters will not impose much overhead. For instance, in the AMD Athlon processors, a single instruction will read a counter register and place the result in a general purpose register [AMD02]. A few additional instructions would be needed to store and compare instruction counts.

Since there might be a significant amount of time between the prediction and corresponding update, it is not certain that a lookup will return the result of the last invocation of the module; rather, it will be the latest that has finished. In addition, updates might not come in sequential order. However, as will be apparent in the next section, the accuracy of this simple predictor is very good for the thresholds of interest. I note that the design space of implementation of such predictors is large, but it is outside the scope of this thesis to study other alternatives.

4.2.2 Experimental Results

Figure 4.4 shows a comparison between the speedup using oracle-determined run-lengths according to Section 4.1, and the last-outcome predictor described in Section 4.2. In the full graphs, speedup using oracle run-lengths are shown as solid lines, and speedup for the predicted lengths are shown as dotted lines. In the bar chart, grey bars are for oracle results, and black bars prediction results. Prediction accuracy is printed above the bars. Only results for overheads of 200 cycles are shown; results for 100 and 500 cycles are similar in behavior, but the differences smaller and larger in magnitude, respectively.

Overall, the predictor manages to obtain virtually the same speedup as the oracle prediction scheme, with a prediction accuracy typically above 90%. For Go, the last-outcome predictor is much better than the oracle-determined length for a threshold of 5000+. This is because the oracle at the same time disables more modules than the predictor (decreasing parallel coverage), and suffers from an increased number of misspeculations. In this particular case, the imperfection of the last-outcome pre-

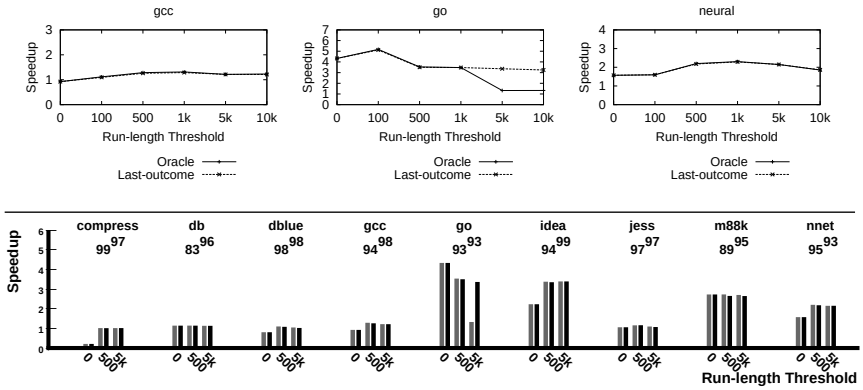


Figure 4.4: *Speedup of the last-outcome run-length predictor (black bars) compared to the ideal predictor (grey bars), with 200-cycle overheads. Prediction accuracy (in %) is printed on top of the bars.*

dictor was beneficial! However, it occurs for a threshold higher than the best. If the predictor happens to fail on threads which are above the best threshold but below the chosen one, it is reasonable that the speedup is better for the predictor than the oracle.

Table 4.2 lists the best found thresholds for all applications at 100 and 200 cycle overheads. The improvement in speedup with the best found threshold is compared to running the programs without module run-length prediction. Note that the improvement for Compress is moot for reasons discussed earlier. The two applications marked '500+' showed a similar speedup for thresholds above 500 and up to 10000, which is the highest threshold used.

Table 4.2: *Speedup improvement.*

App. name	*Best threshold*	*Improvement oh=100*	*Improvement oh=200*
gcc	1k	3%	39%
compress	500+	117%	380%
db	-	0%	0%
dblue	500	8%	34%
go	100	14%	18%
idea	500+	23%	50%
jess	100	4%	7%
m88ksim	100	1.0%	1.6%
neural	1k	17%	46%

In summary, the speedup results at best threshold using the run-length predictor is typically within two percent of the results of an oracle. In addition, with overheads of 200 cycles, six of the nine benchmarks show a speedup improvement, which is between 7-50% compared to running all modules speculatively.

4.3 Systems with Limited Thread Contexts

I have shown that module run-length prediction is useful for preventing the creation of speculative threads that will not contribute to speedup. In this section, I show that the same technique can be beneficial for speculation systems where the number of thread contexts are limited, as discussed in Section 3.4.4.

4.3.1 Experimental Results

For some of the programs, when the maximum number of thread contexts is low, speedup suffers significantly – all finished and preempted threads cannot be kept in the system until they can commit. The programs that benefit from run-length prediction with limited contexts are Go and M88ksim, shown in Figure 4.5. The other applications are not affected much, since misspeculations is the major problem. As the module run-length threshold is increased, fewer threads are created, and not as many threads need to be kept in the system. Consequently, the problem with limited thread context support is less significant.

The figure shows how speedup, on the vertical axis, vary with different run-length thresholds, on the horizontal axis. The lines show result for 8, 16, 256, 1024, and an infinite number of thread contexts. It is obvious that, for the experiments with few available contexts, the speedup is significantly improved with run-length prediction.

With better value prediction or in programs with fewer misspeculations, this technique could be even more important; in simulations with perfect value prediction,

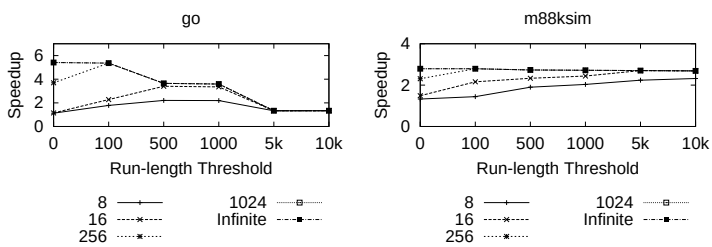


Figure 4.5: *Benefit of run-length thresholds with limited thread contexts.*

seven of the applications benefit from run-length thresholds when the number of thread contexts are limited.

The best threshold may be different from what is reported in the previous section. For example, Go with a maximum of 8 or 16 threads performs best at a threshold of 500-1000, compared to the best threshold of 100 found in Section 4.2.2. The combined effect of thread context limit and overheads should be considered when choosing threshold for such a system.

4.4 Related Work

The Hydra TLS project [HWO98] used thread timers to prevent too short or too long threads to be spawned. Their experience is that 300-3000 instructions per thread is optimal. There is no closer description how these timers work.

Vijaykumar and Sohi [VS98] describes how to choose tasks for multiscalar processors. One criteria is task size; their heuristic is that tasks should be at least 30 instructions. Since task start overhead is two cycles in this architecture, this means the overhead is about 6% of a task with 30 instructions, which they find acceptable. For Multiscalar processors, the tasks are defined by a compiler. Thus, as opposed to the dynamic technique presented in this chapter, their method is implemented in the compiler. There are also other TLS compilers which use thread size as a criterion for thread selection [TJY99, LTS+06].

The DMT architecture [AD98] use thread size as a criteria for spawning new threads. They use a saturating two-bit predictor which depending on thread size, overlap, and retirement determine if a thread should be used for speculation. Again, no further details are provided.

Zahran and Franklin [ZF03] propose *dynamic thread resizing*, a technique to merge or split statically created tasks at run-time in order to adjust the thread granularity dynamically.

4.5 Conclusions

I have presented a new technique for reducing the impact of thread-management overhead in speculative module-level parallelism. I use the *module run-length* to determine if a new thread is to be created for the call continuation. If the run-length exceeds a certain *run-length threshold*, a new thread is created; otherwise the code is run sequentially. Empirically, I have found that 500 cycles is a good threshold for overheads in the range 100-200 cycles.

Module run-lengths are not known until the module has completed execution, but the decision to speculate must be made when the module is called. I have solved this

with a module run-length predictor, which stores whether the run-length was above or below the threshold. The most recent result is used as a prediction for the next invocation of the same module.

The last-outcome predictor is shown to have a very good accuracy, between 83% and 99% compared to an oracle. In addition, six of the nine benchmarks show a speedup improvement when using run-length prediction. For overheads of 200 cycles, the improvements range from 7% to 50% compared to running all modules speculatively.

5
Parallel Overlap Prediction

The drawback with aggressive module-level speculation, i.e. creating speculative threads for all module continuations, is that the overhead can easily dominate the execution time, preventing us from achieving the best possible speedups, and in some cases even cause slowdown compared to sequential execution.

Overhead is defined as all extra work associated with events that do not occur in the sequential execution of the program. One type of overhead is all the work the speculation system performs in order to manage speculative threads. This thread-management overhead consists of thread-start, roll-back, and commit (thread completion) overhead. Some overhead is compulsory and equal for all threads. Its impact is therefore proportional to the size of the overhead compared to the size of the speculative threads – thread-start and commit belongs to this category. Another type of overhead is related to the actual program being executed. When threads are rolled back due to a misspeculation, the work done by the squashed threads is thrown away, this is execution overhead.

Overhead related to misspeculations does, in contrast to the compulsory overhead, not contribute to the sought-after parallelism. This overhead is harmful for several reasons. First, threads that are squashed will have occupied processing and communication resources, as well as storage space for its speculative state, without contributing to the forward progress of the program. In fact, this can potentially hamper successful threads, thereby slowing down execution. Second, even if there are

plenty of free resources, overhead is a serious drawback when considering energy-efficiency or the resource usage in a multiprogrammed environment.

In this chapter I consider how to *predict parallel overlap*, or the time a thread and its speculative child execute in parallel. Threads that roll back well into their execution are the worst offenders, since more work is squashed the longer the thread has executed, and the possibility to still do some useful parallel work decreases. The rationale behind this method is that if the *parallel overlap* – the time a thread and its child executes simultaneously – is sufficiently small, it is either because the module was small, *or* because the child was recently restarted after a misspeculation. Therefore, I apply a minimum threshold for the overlap; call instructions where the overlap is found to fall below the threshold are classified as non-parallel and prediction is used in an attempt to avoid spawning new threads at these calls for future invocations.

While the profiling run shows that seven of nine applications improve speedup, a real predictor does not reach the same success. Still, the total overhead, that is the sum of all types of overhead from all used processors, is brought down to half the original amount using the parallel overlap prediction technique.

5.1 Speculation Overhead

Since the mechanism presented in this chapter aims at reducing overhead, a closer look at the different sources of overhead is needed. As mentioned, all extra work that do not occur in a sequential execution of the program is defined as overhead and is harmful for several reasons.

The *thread-management overhead* is compulsory and consists of thread-start and commit, the time it takes to start and finalize a thread respectively. The compulsory overhead is needed to extract thread-level parallelism. If, for instance, thread-start is time-consuming, the benefit from starting a new thread in terms of parallel execution of useful code will suffer. The impact can be kept under control with efficient speculation mechanisms and by avoiding to create too small threads. In Chapter 4, I presented a method that will prevent small modules from being used for speculation.

I also include roll-back overhead in the general category of thread-management overheads, since it is controlled by the speculation system. However, it is not compulsory. Roll-backs occur as a result of a misspeculation, and thus can be avoided if a method is found to avoid the dependence violation that caused the misspeculation.

In addition to the roll-back handler overhead, partially completed threads that were squashed due to a misspeculation need to be re-executed. The *execution overhead* is work done by threads that had to be squashed. A related type is *communication overhead*, or the extra memory accesses and intra-processor communication caused by threads that are later squashed. Communication overhead could be seen

as part of the execution overhead, but deserves special mention since it potentially hampers other threads competing for the same resources.

Figure 5.1 shows two example execution snapshots. To the left, a successful speculative thread with only compulsory overhead in the form of a thread-start (commit is omitted in the figure to reduce the clutter, but is located at the end of every successfully finished thread). In the example to the right, however, the speculative thread suffered from a dependence violation and had to be restarted. In addition to thread-start, there are also execution-, roll-back, and communication overheads.

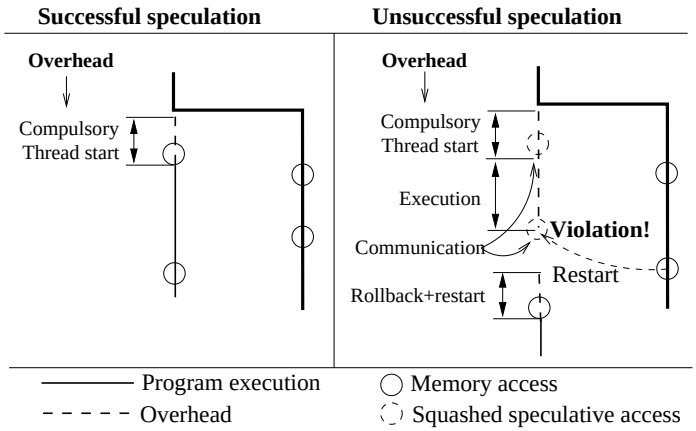


Figure 5.1: *Sources of overhead in thread-level speculation.*

In summary: *Total overhead = thread-start + commit + roll-back + execution + communication.* Throughout the rest of the thesis, I refer to the total overhead unless otherwise stated. The simulation results measure the total overhead as the percentage of extra cycles incurred by all forms of overhead added together compared to the number of cycles used in a sequential execution of the application.

5.2 Parallel Overlap

As mentioned, parallel overlap is the time a thread and its child executes simultaneously, or in other words, the amount of time that useful parallelism is uncovered by spawning a new thread. The overlap is measured and if it is found to be lower than a certain threshold, it is considered undesirable to spawn new threads for the module continuation at future invocations of the call. Therefore, the potential spawn point is classified as non-parallel. The parallel overlap predictor will try to record all non-parallel thread spawn points and make sure no new threads are spawned at those

calls.

If the parallel overlap is smaller than the thread-start time, spawning a new thread will not contribute any useful parallel work. As a result, it makes sense to have a threshold at least as large as the thread-start overhead, which will weed out both misspeculating and too small modules. Larger thresholds may be useful if the misspeculating modules are the ones disabled, but useful parallelism can be removed in the process.

An advantage of this technique is that threads that suffer from a misspeculation early in their execution are not classified as non-parallel. If a thread rolls back early, less work has been wasted, and there is a good chance that it will still contribute with useful parallel work when restarted. The potential drawback is that the total overhead can still be large, since these misspeculations are allowed to occur.

This technique is related to the run-length prediction technique from chapter 4 in that a threshold is used to assess the amount of parallel overlap between threads, but the way overlap is measured and used differs. Run-length prediction measures the length of modules and predicts them to be non-parallel if the length is below a certain threshold. The module run-length includes modules run sequentially within a module, but excludes all overhead and idle (preempted) time. The aim of run-length prediction is to find threads of suitable length for speculation, not primarily to reduce the amount of misspeculations. A big advantage is that the run-length can be measured even when speculation is disabled, so the technique can easily adapt to changing circumstances during execution.

The parallel overlap prediction technique is affected by misspeculations, which is important as the goal is to reduce overhead. It is also simpler to implement. However, once the prediction not to speculate at a spawn point has been made, the ability to measure overlap and thus to re-evaluate that decision is lost. This since new threads will no longer be spawned at the affected spawn point.

5.3 Algorithm & Implementation

Figure 5.2 shows how the parallel overlap is measured. When a thread T1 completes its execution, the start or latest restart time of the most recently spawned child thread T2 is checked and the difference between the child start time and current time is the execution overlap between these two threads.

If the start time is far enough in the past, as in 5.2 (a), the execution overlap will exceed the threshold and the speculation is classified as successful. If, on the other hand, the thread is squashed and restarted late as in Figure 5.2 (b), the execution overlap falls below the threshold, and the child thread is classified as undesirable. When this happens, a prediction mechanism is used that aim at preventing the same thing from happening again. Because of the simple way to measure overlap, this

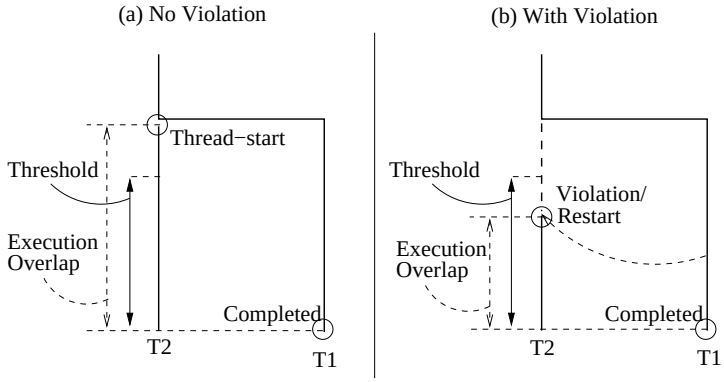


Figure 5.2: *Calls are marked as non-parallel if the parallel overlap is below the threshold.*

technique is only suitable for thread roll-back where the whole thread is restarted after a misspeculation.

Implementing this technique requires bookkeeping of the start time of each active thread, as well as a table that predicts whether or not to speculate. One prediction per module is stored, and index the prediction table with a module ID. The ID could be any identifier unique for the module, such as a sequence number or the address of the first instruction. As is shown in the rightmost column in Table 3.2 there are only up to a few hundred modules in the applications. Therefore, if only one prediction per module is stored, the prediction table does not need to be larger than a few hundred bytes to avoid having several modules map to the same slot. In the simulations an infinite prediction table is assumed, which is not unreasonable due to the small number of modules. If the table is stored in the memory hierarchy, it is automatically shared between the processors.

The extra work needed for this algorithm consists of:

- Record start time for each thread. The speculation system needs to keep a list of active threads; the start time – this can be a cycle count obtained from the processors' built-in performance counters – is recorded at thread-start.

- When a call instruction is encountered, the prediction table is accessed; if the prediction is *speculate* a new thread is created, otherwise the speculation system does nothing. If the start address of the module is used as module ID the index to the prediction table is the same as the target address of the call instruction.

- When a thread is completed, the start time of the closest more speculative thread is read and compared to the current time; if the difference is below the

threshold a *no-speculate* prediction is written to the prediction table, otherwise nothing needs to be done.

These operations can be added to thread-start and completion operations in the speculation system with only a few extra instructions and memory accesses – which is not much compared to a full thread-start. Since the overhead for the overlap prediction operations is expected to be small compared to the existing overheads, no extra time is added for them in the simulations.

5.4 Simulation Methodology

The simulation setup and benchmarks are the ones described in Section 3.2. The simulator is extended with the algorithm for predicting parallel overlap. Some parameters used in the experiments are listed in Table 5.1.

Table 5.1: *Baseline machine parameters - parallel overlap prediction.*

Feature	*Baseline parameters*
Processors	8
Overhead	100 cycles for thread start, roll-back, commit, and context switch.
Rollback policy	Thread roll-back.
Value prediction	Stride return value prediction. No memory value prediction.
Parallel overlap threshold	100, 120, or 200 cycles.

5.5 Experimental Results

In order to evaluate the potential of predicting parallel overlap, I first run simulations with profiling; the application is executed once while call instructions where overlap is below the threshold are marked as non-parallel. The same workload is then re-executed with speculation disabled for these calls. Note that this does not necessarily give an exact result according to my definition of parallel overlap – as soon as one thread is removed the rest of the execution will change, which is not taken into account in the profiling run – but it will give a fair estimate of the effectiveness of the method. The results from the profiling run will then be compared to a realistic implementation of parallel overlap prediction.

5.5.1 Parallel Overlap Profiling Results

Figure 5.3 shows speedup results and total overhead for the applications with overlap thresholds of 100, 150, and 200 cycles. The lowest threshold, 100, is equal to the thread-start overhead, and will therefore remove threads that do not overlap enough to account for the thread-start procedure. As a reference, results for thread roll-back from Figure 3.11 are included, they are labeled *all* since all calls will spawn a new thread.

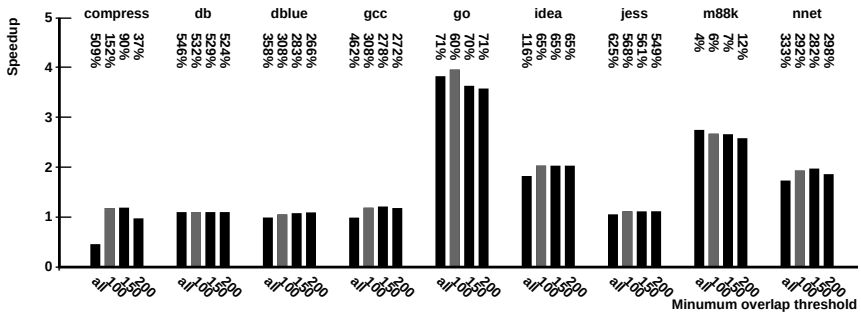


Figure 5.3: *Disable speculation if overlap is less than 100, 150, or 200 cycles.*

An overlap of 200 cycles may not seem to be much, but one can already see how speedup starts to decline for several of the benchmarks (Gcc, Compress, Go, M88ksim, Neuralnet), which suggests that higher thresholds are not useful. On the other hand, a threshold of 100-150 is beneficial for most of the programs; only Db shows virtually no improvement, and M88ksim a small decrease in speedup (M88ksim is an exception, where using this technique happened to cause new dependence violations and increased squashing). However, the drawback of the technique is that the total overhead is still large for many of the programs. For instance, Gcc requires almost three times as many clock cycles as the sequential execution in order to achieve a speedup of about 25%.

The threshold that yields the best speedups, highlighted in grey, is the one equal to the thread-start overhead of 100 cycles. For this threshold, the average overhead for all applications is down to 255% compared to 336% when starting all modules speculatively (the leftmost bar).

5.5.2 Parallel Overlap Prediction Results

Figure 5.4 shows, from left to right, the all speculative and profiling results carried over from Figure 5.3, compared against new results from a predictor working at run-

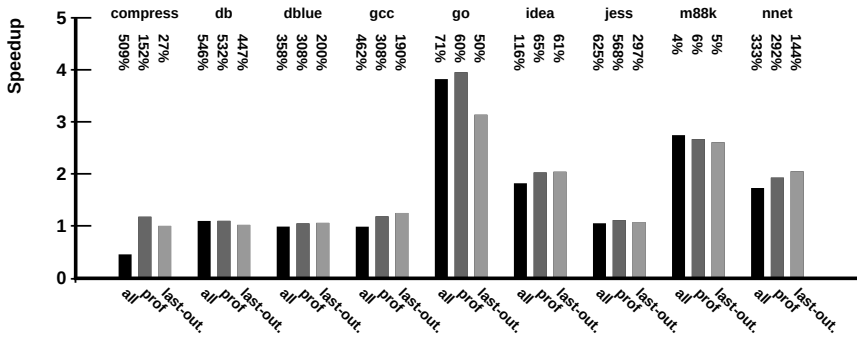


Figure 5.4: *Last-outcome parallel overlap prediction compared to the profiling results.*

time. It is a per-module, last-outcome predictor updated at thread completion. Both profiling and predictor results are with a 100-cycle overlap threshold.

Overall, the predictor is somewhat over-zealous in disabling speculation compared to the profiling run, which can be seen on the lower numbers for total overhead (mean 158%). This translates into lower speedup for Compress, Go, and M88ksim, slightly lower for Db and Jess, but slightly better for Gcc and Neuralnet. Especially for Go, however, the predictor is not performing well, and the overhead is still significant in many of the applications.

The results show that while the simple predictor manages to reduce the overhead, this is to the expense of losing parallelism exploitation opportunities in comparison with the profiling method. In addition, the overhead remains high; on average 255% and 158% for the profiler and the predictor, respectively, with five of the applications having in excess of 200% overhead. The speedup is improved for seven of the applications in the profiling run, but only five with the predictor, and three applications performed worse using the technique. A more advanced predictor might do better; however, investigating alternative predictors is beyond the scope of this thesis.

5.6 Related Work

The DMT architecture [AD98] use parallel overlap as one criteria for spawning new threads. A two-bit saturating counter for each thread spawn point predicts if a thread will perform well. Overlap and two other criteria are used. My results have not been compared to the DMT overlap prediction technique since no details of their implementation are provided, and results for overlap prediction in isolation are not presented.

5.7 Conclusions

In this chapter I proposed a technique to use the *parallel overlap*, or the amount of time a thread executes in parallel with its child thread, as a way to filter out the most harmful misspeculations. The *parallel overlap predictor* records potential thread spawn points where the measured overlap has previously found to be below a certain threshold; for my system a threshold of 100-150 is found to work well. No new speculative threads are spawned at these potential spawn points.

The speedup is improved for five of the nine benchmarks, and the average overhead reduced to less than half compared to indiscriminate speculation, from 336% to 158%. While the results are encouraging and show that parallel overlap prediction indeed both reduces overhead and improves speedup for most applications, the remaining overhead is still substantial.

This chapter is a revised version of the previously published paper "Reducing Misspeculation Overhead for Module-Level Speculative Execution" [WS05].

6 Misspeculation Prediction

In this chapter I propose *misspeculation prediction*, a technique aimed at avoiding to spawn threads that will misspeculate. The method uses history-based prediction, that is prediction based on previous violations, when deciding whether to create a new thread or not. It can be integrated with a speculation run-time system. The goal is twofold: to reduce the total overhead, and if possible improve speedup compared to naively spawning new speculative threads for all module calls.

I investigate a number of predictors and different ways to record misspeculations, and find that using a simple last-outcome predictor indexed with a module identification number can bring down the overhead a factor of six compared to indiscriminate speculation. Speedup is improved for four applications, but noticeable worse for two applications.

The applications that do not benefit from misspeculation prediction are those who suffer from few misspeculations to begin with. By applying misspeculation prediction selectively, i.e. only when the ratio of squashes compared to new thread starts is above a certain threshold (0.6 works well for my applications), the negative impact on speedup is avoided at the expense of slightly higher overhead. I find that this method gives the same or slightly higher speedup for all applications compared to indiscriminate module-level speculation, but with almost four times lower average overhead.

There have been other attempts to avoid misspeculations. Several techniques

record cross-thread dependences and synchronize dependent load-store pairs [CT02, MS97, SCZM02]. However, the threads will still incur thread-start and commit overheads. In contrast, threads that are expected to misspeculate will not be created with my approach, which means thread-start and commit overhead is avoided.

This technique is an alternative to, or could possibly work in conjunction with, the *parallel overlap prediction* technique presented in Chapter 5. The goals of the two techniques are similar. The main difference is that the parallel overlap prediction technique will only predict spawn points where misspeculations have occurred late in the dependent thread as non-parallel, while misspeculation prediction more aggressively disables speculation for spawn points where misspeculations have occurred. Therefore, the misspeculation prediction technique is more successful at reducing the total overhead incurred by thread-level speculation.

Section 6.1 introduces the misspeculation prediction technique, and in Section 6.2 a number of implementation design choices are investigated. Then, misspeculation prediction is improved upon with a method for selective use in Section 6.3. Related works are discussed in Section 6.4, and finally, I conclude in Section 6.5.

6.1 Predicting Misspeculations

The misspeculation prediction technique attempts to selectively disable speculation whenever a thread causes misspeculations. Every time there is a dependence violation, the call tree is analyzed in order to find a confluence point where, if a new speculative thread is not spawned, the dependence violation will disappear. Once the confluence point is identified, I propose to use history-based prediction in order to avoid expected misspeculations when the same situation occurs again.

The main advantage of this method is that all misspeculating threads are targeted while successful threads are left alone, which ought to ensure a significant reduction in misspeculations and therefore low overhead. A drawback is that threads which misspeculate early and later contribute with useful parallelism are also affected; the technique does not search for the best tradeoff for maximum speedup.

I describe the method in Section 6.1.1. The impact on speedup and overhead is then investigated in Section 6.1.3.

6.1.1 Algorithm & Implementation

A first concern is which call instruction to classify as non-parallel after a violation. In fact, there are several possibilities to select module(s) as non-parallel in order to get rid of the misspeculation. In Figure 6.1, there is a dependence violation between a store in thread T1 and a load in T4. In order to avoid the violation, one must make sure the store is executed before the load. To achieve this, one or several of the confluence

points (*A*, *B*, and *C* in the figure) that define the relative position of these instructions must be selected.

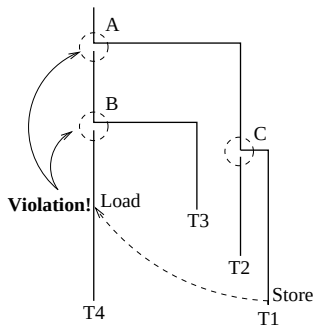


Figure 6.1: *Finding calls to classify as non-parallel after a violation.*

Choosing either *A* or *B* alone might do the trick, since both will delay the execution of the load. Which one is better to select will depend on the situation. Intuitively, a good heuristic could be to select *A*, the common ancestor for the threads involved in the violation, as being more likely to delay the load sufficiently. The advantage of *B*, however, is that it is the confluence point where the misspeculating thread was created; thus it will be easy to find the right call instruction to mark non-parallel in conjunction with the roll-back operation.

Only *A* and *C* together is certain to remove the violation, serializing all code from the common ancestor to the problematic store instruction. However, this is undesirable for several reasons: first, we do not want to remove more parallelism than necessary to avoid the violation; second, finding and inserting multiple calls in the prediction table would take more time; and finally, if the first attempt failed, the predictor will add another non-parallel prediction in order to get rid of the violation if the situation repeats. I will experiment with both the closest fork (called type *B*), and the common ancestor (type *A*) predictors.

Implementation of this technique requires that the roll-back handler is augmented to find type *A* or *B* confluence points based on the knowledge that T1 and T4 contains the conflicting instruction pair. The relevant confluence points can be found using the list of threads active in the speculation system; this list is already accessed by the roll-back mechanism in the course of squashing, so the overhead should be small compared to the existing mechanism. A prediction table is also needed; it will be accessed at module calls and updated during roll-backs if needed. The prediction table will be discussed in the next section.

6.1.2 Simulation Methodology

The simulation setup and benchmarks are the ones described in Section 3.2. The simulator is extended with the various possible misspeculation prediction techniques. Some parameters used in the experiments are listed in Table 6.1.

Table 6.1: *Baseline machine parameters - misspeculation prediction.*

Feature	*Baseline parameters*
Processors	8
Overhead	Baseline is 100 cycles for thread start,
	roll-back, commit, and context switch.
	10- and 50-cycle overheads used in one experiment
Rollback policy	Thread roll-back.
Value prediction	Stride return value prediction.
	No memory value prediction.
Prediction table size	Baseline is infinite table.
	256 or 1024 entries used in one experiment.

6.1.3 Experimental Results

I begin with profiling experiments. The workload is run several times, each time all type *A* or *B* calls that cause a misspeculation are marked non-parallel when re-executing the same workload. This process is repeated until there are no longer any misspeculations or a maximum of five iterations, lest too much time is spent on finding and removing the last few misspeculations. While the profiling method is approximate, it will still give a fair estimate of the potential for using misspeculation prediction.

In Figure 6.2, the leftmost bar for each application, marked *all*, is the naive implementation of running all modules speculatively. The next two bars, *prof/A* and *prof/B*, show results for the profiling runs for type A and B confluence points respectively. For each application, the height of the bars indicate speedup on the speculative CMP compared to sequential execution. The numbers on top of the speedup bars show the total overhead as described in Section 5.1.

The average overhead for indiscriminate speculation is as hight as 336%. Even programs with decent speedup can have high overhead, for instance, Neuralnet with a speedup of almost 2 and 333% overhead. This is possible since the 8-way machine has capacity to get useful work done even if some threads are repeatedly squashed and re-executed.

As expected, the total overhead is greatly reduced. The average overhead for type A is 41% and for type B 28%; in no case does the overhead exceed 100% of the

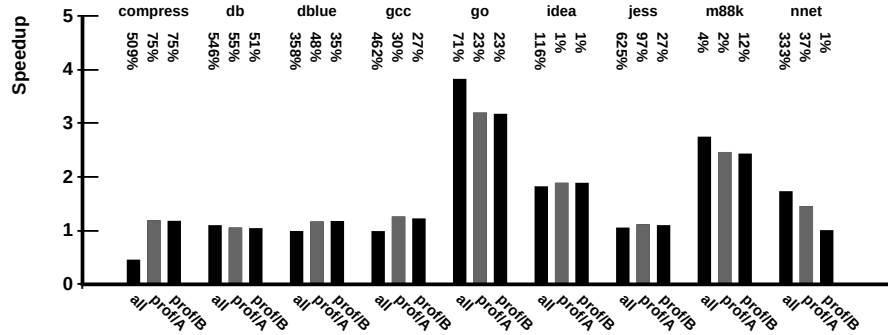


Figure 6.2: *Profiling results for disabling speculation based on misspeculations.*

serial execution time. The lower average overhead for type B can be attributed to lower overhead figures for Jess and Neuralnet. However, this is at the expense of worse speedup, especially for Neuralnet which is almost serialized with *prof/B*. In addition, with type B there are actually more threads squashed than in indiscriminate speculation for M88ksim, resulting in the higher overhead figure. For these reasons type A, highlighted in grey, seems to be the better choice. Speedup improves for five programs, but is reduced in four cases (although marginally for Db) compared to running everything speculatively.

In summary, by removing misspeculating threads with profiling, the overhead can be brought down from an average of 336% to an average of 41% or 28% while at the same time improving speedup for about half of the applications. While this is encouraging, in the next section I will investigate how well a predictor can exploit this potential. However, there is the risk of disabling speculation even where useful parallelism do exist. This happens in a few of the benchmarks, notably Go, M88ksim, and Neuralnet. This problem, and a potential remedy, will be discussed in Section 6.3.

6.2 Design Space for Misspeculation Predictors

The misspeculation prediction technique shows promise in bringing down the overhead in module-level speculation. This section contains a thorough investigation of the design space. I begin with a discussion of the design space before analyzing the performance of a number of designs.

6.2.1 Predictors & Implementation

Once unwanted threads are identified, a prediction table is used to store information about the misspeculations. The *indexing method* determines how future events are matched and identified as probable misspeculations. The goal is to catch future instances where the same module is called and yet another misspeculation expected. However, the same module can be called from multiple places in the code. This means an entry in the prediction table could cover everything from only one of those calls to all of them, depending on how we chose the prediction table index. Having a single entry cover multiple call places, for instance, saves space in the prediction table, and the warm-up time will be shorter. On the other hand, prediction accuracy might suffer.

Some interesting options are:

- Per-call: The prediction table is indexed with the call instruction address; i.e. the table entry only covers a single call instruction in the application.
- Caller/Callee: A concatenation of the module IDs for caller and callee modules: the scope is expanded to cover similar situations in the same module (repeated calls to the same function).
- Callee only: Index by module ID, the same table entry is used regardless of where the module was called from.

The module ID could be any identifier unique for the module, such as a sequence number or the address of the first instruction. I have not evaluated the per-call option due to limitations in the simulator, but the latter two are evaluated and compared in the next section.

The next question is which *predictor* to use. I begin with a simple last-outcome predictor, which will disable speculation on a module as soon as a violation has occurred. In order to avoid making a decision based on an exceptional case, the last-outcome predictor can be enhanced to an n-bit saturating counter type predictor. Experiments were run with last-outcome and 2-bit predictors. For the 2-bit predictor, speculation is disabled when the high bit is set, i.e. after two consecutive misspeculations.

A disadvantage of the misspeculation prediction technique is that it lacks the ability to re-evaluate a no-speculate prediction. Once speculation is disabled, it can no longer detect if circumstances change since future invocations will be run sequentially. The last point in the design space is *prediction duration*. The prediction can either be permanent, or it can time out with some interval in order to make a reevaluation. I will investigate having the prediction time out and reset to zero after it has been accessed k times.

The implementation issues are the same as those described in Section 6.1.1, however, the prediction table was not discussed. The table should be shared among the cores in the CMP. It could be stored in memory, where it is automatically shared between the processors. Since the table is updated only when a prediction changes – at first misspeculation or timeout – most accesses will be read-only, which should help keeping sharing overhead due to invalidations relatively low. Preferably the table should not take up too much space. As is shown in the rightmost column in Table 3.2, there are only up to a few hundred modules in the applications. Therefore, if one prediction per module ID is stored, the table does not need to contain more than a few hundred entries to avoid having several modules map to the same slot. The other options will require somewhat larger tables to avoid interference. In most of the simulations an infinite prediction table is assumed, which is not unreasonable due to the small number of modules; however, an experiments with finite table size has also been run to confirm this supposition. The predictions use one or two bits each, plus an expiration counter when timeout is used. For the ranges of interest, a six-bit timeout counter would suffice.

6.2.2 Experimental Results

Impact of prediction table indexing method

In Figure 6.3, three ways to store a prediction are compared, The *Callee/A* and *Callee/B* bars show speedup when storing predictions based on callee module ID, and choosing the module of type *A* or *B* respectively. The rightmost bar, labeled *Caller+Callee/A*, instead uses a concatenation of the caller and callee module IDs as index for the prediction table. The *all* and *prof/A* results are carried over from the previous section for comparison. In these simulations, a last-outcome predictor is used.

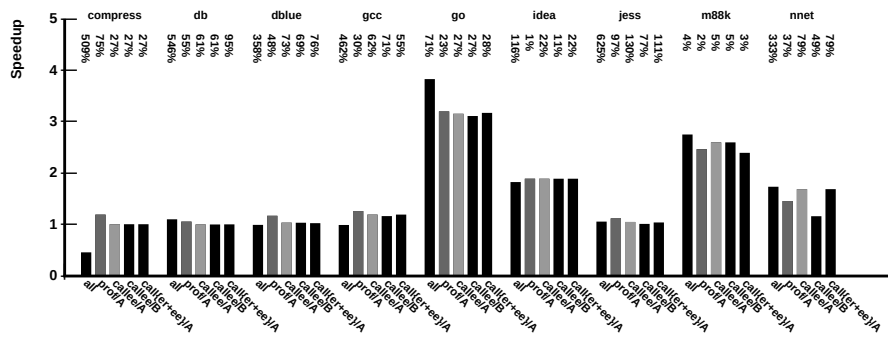


Figure 6.3: *Comparison of misspeculation prediction policies.*

It can be seen that the methods perform similarly in most cases, both with respect to overhead and speedup. Only for M88ksim does the *Caller+Callee/A* give a lower speedup than the others, and the same is true for Neuralnet and *Callee/B*. One can conclude that the increased resolution of caller+callee IDs does not improve the result. Since the simpler module ID indexing method is more space-efficient, I rule out the caller+callee option. *Callee/A* seems to consistently yield the best results, which conforms with the profiling results.

The predictor generally runs more modules speculatively than the profiler, with Compress being the exception. Not all misspeculating modules are correctly predicted as such; hence the predictor shows somewhat higher overhead figures, and in most cases lower speedup. However, for M88ksim and Neuralnet the speedup is better with more speculation; some parallelism is lost when disabling misspeculating modules in these applications.

Impact of choice of predictor

Based on the choice of *Callee/A*, I proceed to examine three different predictors, the last-outcome, a 2-bit predictor, and a 2-bit predictor with timeout. The results are shown in Figure 6.4. Timeout is set so that a prediction expires after it has been accessed 20 times. Timeouts in the range of 10-100 accesses were investigated, with 20 showing the best overall result.

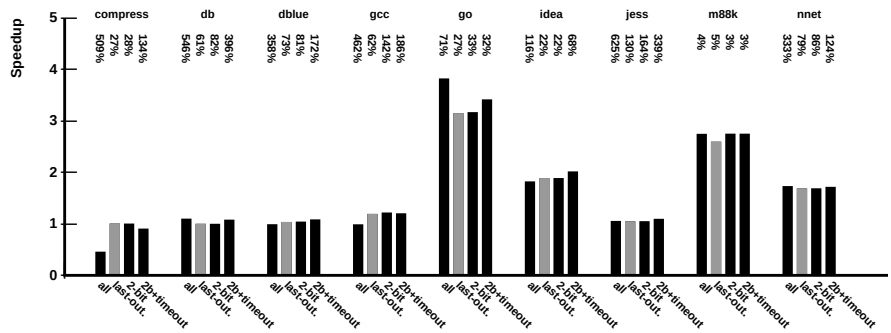


Figure 6.4: *Performance of the last-outcome, 2-bit and 2-bit + timeout predictors.*

It is clear that last-outcome and 2-bit predictors perform virtually the same, only for M88ksim is there a slight improvement from using the 2-bit predictor. However, because the 2-bit predictor takes longer before reaching the decision to disable speculation, the number of misspeculations and consequently the total overhead is generally somewhat larger; significantly larger in the case of Gcc. With timeout added, the difference in overhead is even more pronounced. Some programs benefit from the

timeout, namely Go, Idea, and Db, but the overhead of Db also increases from 82% to 396% with the timeout enabled. The average overhead is 54% with a last-outcome predictor, not unreasonably higher than the 41% reported by the profiling run. The 2-bit predictor has a slightly higher 64% average overhead, and with timeout the results is a significantly higher 161%.

Impact of number of processors and speculation overhead

In order to make sure the last-outcome misspeculation predictor is beneficial with a range of hardware organizations, the experiment were run on a smaller 4-way machine. The results for the 4-way machine are shown in Figure 6.5. In Figure 6.6 an 8-way machine is used once again, but with lower 10- or 50-cycle speculation overheads.

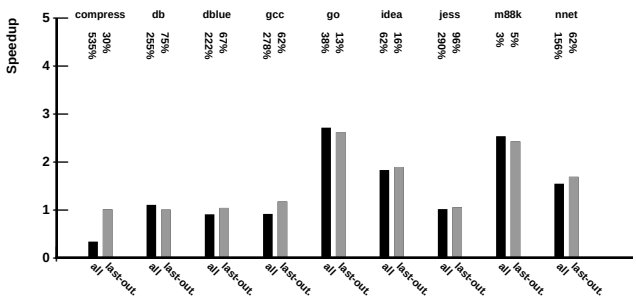


Figure 6.5: *Performance of last-outcome misspeculation predictor with a 4-way CMP.*

As expected, the total overhead is lower on the 4-way machine than the 8-way. Since fewer processors are available, fewer threads can run simultaneously, and consequently less work is squashed due to violations. However, there is still an average 204% total overhead when starting speculative threads for all continuations, compared to an average 47% when employing the technique. The impact on speedup is very similar to what can be seen for the eight-way configuration.

The second variation is the speculation overhead, since it is not known exactly how much overhead will be imposed in a real implementation. The technique is less likely to yield good results on machines with low speculation overheads; therefore, experiments were run with 10- and 50-cycle overheads. As is evident in Figure 6.6, the misspeculation predictor still produces good results. With 50-cycle overheads, the average total overhead goes down from 274% to 53% with my technique. With 10-cycle overheads, the average decreases from 230% to 51%. The total overhead does not decrease as much as one might think with lower speculation overheads.

This is due to the fact that the number of misspeculations increases when threads are started in a faster pace. With lower speculation overheads, however, speedup is less affected by misspeculations. Hence, one can see that speedup suffers somewhat for all programs with misspeculation prediction enabled when overheads are as low as 10 cycles.

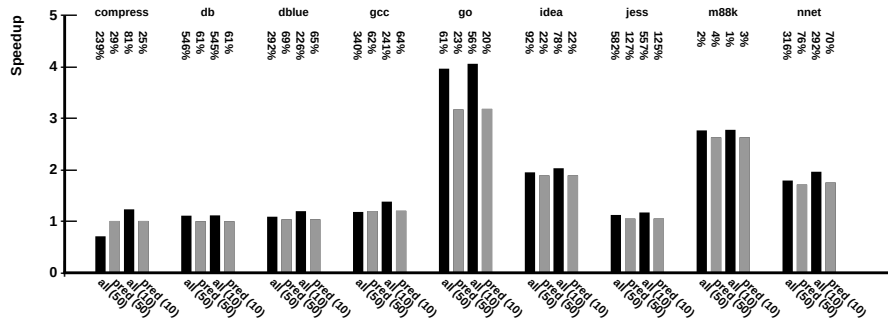


Figure 6.6: *Last-outcome misspeculation predictor with 10- and 50-cycle overheads.*

Impact of finite size prediction table

All previous results are with an unlimited prediction table, i.e. there are never interference between two modules with a different ID. Since the number of different modules is not huge (see Table 3.2) one can expect that a relatively small predictor table will be sufficient for the misspeculation predictor. Figure 6.7 shows results with finite prediction tables of 1024 and 256 entries. Each entry is a single bit containing the last-outcome predictor. The leftmost bar shows the unlimited table used in previous Figures, while the other two bars show results with 1024-entry and 256-entry tables. The tables are indexed with 10 and 8 bits from a 32-bit module-ID respectively. It can be seen that the performance is close to that of the unlimited predictor. When there is some interference, such as for Gcc, the result is that a few modules that should have been run speculatively are instead run sequentially. The overhead goes down somewhat, but at the expense of lower speedup. With a 1024-entry table, four of the nine benchmarks perform identical to the unlimited predictor, and the impact on the remaining five is small.

In summary, the last-outcome predictor with the *Callee/A* table, highlighted in grey in Figure 6.4, seems to be the best choice, yielding a slight speedup improvement on four programs, and the same speedup on two, but with a significantly lower 54% average overhead. Even if the architecture in terms of number of processors or size of overhead changes, the gain achieved with misspeculation prediction remains

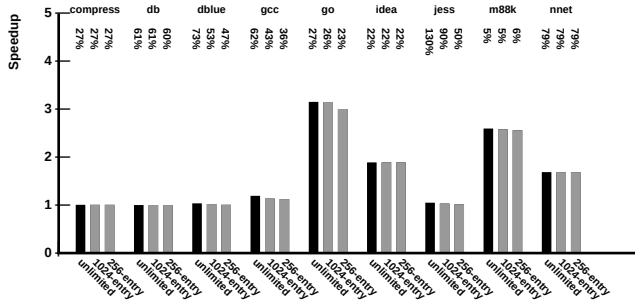


Figure 6.7: *Performance with realistic 256- and 1024-entry prediction tables.*

significant. In addition, the gain can be achieved with a relatively small prediction table.

However, a couple of the programs, Go and m88ksim, works better without the technique enabled at all. The overhead is small to begin with, and using misspeculation prediction removes useful parallelism and increases the execution time. In the next section I will look at a possible solution for that problem.

6.3 Selective Use of Misspeculation Prediction

The results from the previous sections show that misspeculation prediction is an efficient way to reduce the misspeculation overhead while achieving the same or slightly higher speedups as running everything speculatively. However, a couple of applications, Go and M88ksim, did not benefit from the technique. On the contrary, their speedups are negatively affected. These two programs show good speedups and low overhead without applying a misspeculation reducing technique – there are few misspeculations in these applications to begin with.

In this section, I attempt to add a safeguard which will make sure that misspeculation prediction is not applied to programs which do better without. The reasoning is simple – if there are many misspeculations the technique is enabled and the predictor used when deciding if a new thread should be created or not; if misspeculations are relatively few, the prediction table is not used.

6.3.1 Algorithm & Implementation

In order to get a metric of how prevalent misspeculations are in a program, two global counters are maintained: a squash counter is increased every time a thread is squashed, and a thread-start counter is increased every time a new thread is started.

The ratio squash/thread-starts will, at any point in the execution, be a value between 0 and 1 which shows the fraction of the started threads that have been squashed. If there are many misspeculations, the number goes up; if speculation is successful, the number goes down.

The idea is to have misspeculation prediction, as described in the previous section, with predictors being updated throughout the execution time. However, the predictors are only used in the decision of whether to create a new thread or not if the squash/thread-start number is above a threshold. That way, the use of misspeculation prediction will be automatically enabled and disabled as needed during the execution of the program.

Implementation is simple, only the two global counters, increased by the roll-back and thread-start handlers respectively, need to be added.

6.3.2 Experimental Results

In order to find out if there indeed is a useful threshold simulations were run with thresholds in increments of 0.05. In Figure 6.8, the interesting range of threshold values is shown. Some of the applications are well on either side of the threshold. Misspeculation prediction is always enabled for Db, Idea, and Neuralnet, and always disabled for M88ksim, in this range. For Gcc, Dblue, and Jess the overhead steadily increases as the threshold is increased to allow more misspeculations before the predictors are used, but there is yet no change in speedup.

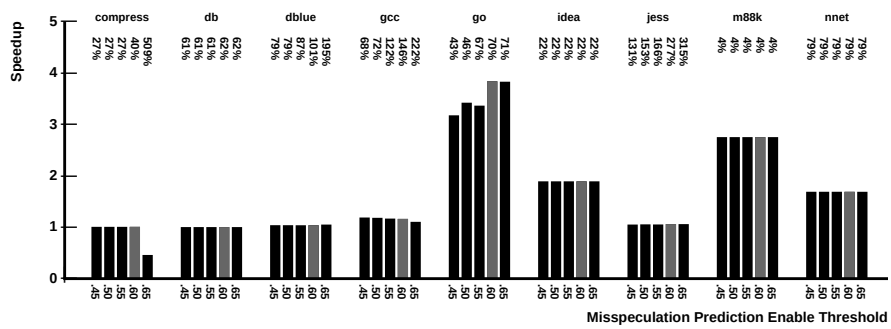


Figure 6.8: *Threshold for misspeculation prediction.*

The sensitive applications are Go and Compress. When the threshold is at 0.65, the speedup for Compress goes down sharply; in fact, there is a large slowdown, due to the fact that misspeculation prediction is permanently disabled. For Go, the opposite is true, when the threshold is 0.55 or lower, misspeculation prediction is

active and removes some useful parallelism. Only a threshold of around 0.6 is fine for all the applications.

With a threshold of 0.6, the average overhead is 89%, up from 54% when using misspeculation prediction without threshold, but with overall better speedup. However, the best threshold is within a rather narrow range, so the potential drawback is that this threshold might not always yield the best results over a larger number of applications. Keeping the threshold as low as possible will at least make sure the overhead is brought down and at the same time decrease the likelihood of suffering from slowdown.

6.4 Related Work

An alternative method to prevent misspeculations is to learn about cross-thread dependences and stall the dependent load until the dependency is resolved. This has been investigated in the context of Multiscalar processors [MS97], speculative chip multiprocessors [SCZM02] and larger DSM machines [CT02] with some success. Even more dependent on synchronization is the superthreaded architecture [TY96], which does not speculate on data dependences at all; instead, inter-thread data dependences are always solved with synchronization, at the price of serialization of all store address calculations.

For the Multiscalar processor [MS97], load-store pairs that are predicted to cause a violation are inserted in a synchronization table. Using this table, dependent loads are stalled until the corresponding store has completed and the value can be forwarded. The technique described by Steffan et al. [SCZM02] is slightly different; a list of violating loads is maintained and when a load that appears on the list is encountered, the thread is stalled until it becomes non-speculative. Finally, the technique by Cintra and Torrellas [CT02] is similar to the one by Steffan: however, they use two levels of stalling: Stall&Release, where the load is stalled until the first writer thread has committed, and if this fails Stall&Wait, which stalls the thread with the load until it is non-speculative. In Hammond et al. [HWO98] they use a simpler synchronization method; the compiler may insert explicit synchronization into the code in the form of a busy-wait loop that reads a lock variable and a store that writes the same lock. My technique differs from these since it tries to avoid creating threads which will misspeculate in the first place; which means that the thread-start overhead is also avoided.

It is mentioned that Hydra uses techniques such as thread timers, stall timers, and violation counters to disable speculation for non-parallel threads and thus decrease overhead [HWO98]. However, neither the implementation nor the performance of these techniques have been reported; thus it is unclear how they relate to my technique.

6.5 Conclusions

When aggressively spawning speculative threads at all module invocations, the execution is dominated by overhead. In my benchmark applications the average overhead is three times as big as the useful work with indiscriminate speculation.

The technique presented in this chapter is aimed at bringing down the overhead in order to save processing and communication resources, as well as reducing the extra energy required for thread-level speculation. The technique can be integrated in the speculation run-time system and does not require recompilation of the programs.

The experimental findings are the following:

- The overhead can be reduced with a factor of six using a last-outcome misspeculation predictor for each module. The speculation system decides whether or not to start a new thread when a module is called based on this prediction. However, the speedup is adversely affected for some applications.

- My technique is shown to work well for a number of chip multiprocessor architectures with varying number of cores and size of speculation overhead. In addition, the technique is shown to work well with a small (in the range of a few hundred entries) shared prediction table.

- When adding a mechanism for dynamically enabling and disabling misspeculation prediction based on whether the ratio of misspeculations to new threads is above a certain threshold (0.6 was found to be the best threshold for the applications used) the average overhead is reduced a factor of four, but with equal or better speedup than indiscriminate speculation for all the benchmark applications.

Overall, this study shows that it is possible to exploit most of the inherent speculative module-level parallelism while removing most of the overhead associated with indiscriminate speculation.

7
A Detailed TLS Model

The chip multiprocessor machine model used in Chapters 3 to 6 is based on a simple single-issue non-pipelined processor core with single-cycle access to memory and remote processor cores. This model was useful for experiments where the aim is to investigate the inherent application parallelism or to look at speculation overhead in isolation from machine dependent overheads.

For machines that are reasonable to build, there are other factors which influence the performance. With a memory hierarchy, memory accesses and communication between threads will be associated with a cost in terms of latency. This will affect the amount of useful TLP, but to what extent and in what way is not obvious. In addition, a modern processor is pipelined and typically employs out-of-order execution in order to take advantage of instruction-level parallelism (ILP). How ILP and and TLP interact is another open question.

Another missing piece in the simulation model used in the previous chapters is the details of the speculation system. While it was demonstrated in Chapter 2 that the major pieces for the assumed TLS support exist in various proposals, there are pros and cons associated with each proposed technique, including limitations that may affect performance.

In this chapter, I will describe a thread-level speculation system and a new simulation framework. The aim of the chapter is two-fold. The first is to describe the methodology and properties of the new simulator used in the remaining chapters. The

motivation for building this new simulator is to move beyond exploration of the inherent parallelism and towards performance improvements one could expect from a real TLS implementation. For that, detailed memory system and processor models are needed. In particular, the aim is to investigate the effect of communication latencies and advanced processors on TLS performance.

The second aim is to make a case that thread-level speculation could actually be implemented in the next generation of microprocessors. While not going into microarchitectural details, the described architecture will contain all the major parts of a TLS architecture. Most of the techniques in this architecture are not new, it leverages techniques in the multitude of existing TLS proposals for CMP and SMT processors, [AD98, CW99a, GVSS98, HWO98, KT99, MG99b, OKP+01, PV03, RTL+05, RSC+05, SM98, SBV95] among others. Wherever applicable, I will refer to these works in the description of the architecture.

It is important to remember, however, that the discussion of a TLS system is focused on the architecture of core TLS functionality. It is beyond the scope of this thesis to specify all details of a working TLS implementation. Therefore, exact mechanisms for thread-start, thread information and prediction table storage and various other issues are not discussed in this chapter.

Finally, in order to broaden my investigation of the TLS design space the new simulation environment will include two major additions. First, loop-level parallelism is added alongside module-level parallelism. Loop-level parallelism is the most popular target among the many proposed TLS architectures. Second, the simulator will include simultaneous multithreaded (SMT) processor cores, in order to facilitate a comparison between how TLS performs on chip multiprocessors and SMT processors.

Guiding principles when making design decisions for the speculation system have been to keep it flexible to allow it to work on different architectures, and to avoid the need for centralized structures.

The chapter begins with an introduction to simultaneous multithreading in Section 7.1, then the implementation of loop-level threads is presented in Section 7.2. Next, in Section 7.3, a multithreaded architecture integrating chip multiprocessing, simultaneous multithreading, and thread-level speculation is described. This is an extensive section detailing a flexible implementation of the functionality needed for thread-level speculation. The run-length and misspeculation prediction techniques are extended to work with loop-level threads. For loops, run-length prediction will also work as a dynamic loop unrolling mechanism. The last section in this chapter, Section 7.4, describes the experimental environment used with the new simulation model. This includes the simulation tool chain, how simulation samples are created from the benchmarks, and a description of the set of benchmarks that are used in subsequent chapters.

7.1 Simultaneous Multithreading

As opposed to chip multiprocessors, SMTs support multiple threads within a single processor core. The aim for SMT processors is to increase the utilization of the execution units, but keep the possibility for threads with a high amount of instruction-level parallelism to run at full-speed. Especially in wide-issue deeply pipelined superscalar cores, the utilization for a single thread can be quite low. This is due to data dependences and even more importantly because threads often stall during long latency memory accesses or after branch mispredictions.

In an SMT processor, several hardware threads run simultaneously on the same core. They share most of the resources in the processor core. Several threads can issue instructions simultaneously, i.e. in the same cycle, provided there are free execution units of the right kind. Figure 7.1 shows the difference between a CMP and an SMT each supporting two threads. Issue slots in the processor are illustrated with boxes. The machine to the left is a chip multiprocessor where each of the two cores can run a single thread and issue up to two instructions per cycle, illustrated by two boxes in each row. Each row shows how many instructions are issued in one clock cycle.

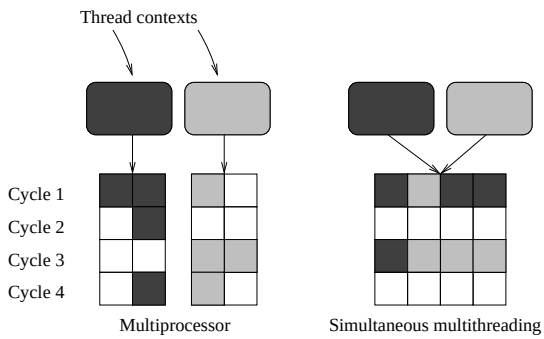


Figure 7.1: *Chip multiprocessor and simultaneous multithreaded cores.*

In the example, there are two running threads. Instructions from one thread are shown with dark squares, and from the other thread with a lighter shade. The dark thread is ready to issue three instructions in the first cycle, followed by a stall cycle after the third instruction, and a final fourth instruction. The other thread issues one instructions, followed by a stall, and then three parallel instructions. In the chip multiprocessor, the instruction-level parallelism is restricted to two, so even if three instructions could be issued in parallel, only two can issue in the first cycle, one is delayed to the second cycle, the third cycle is the stall, and in the fourth cycle the final instruction is issued. The result for the other thread is similar.

The SMT processor uses a wider core and can issue four instructions each clock

cycle, but two threads share the execution units. In the first cycle, all three ready instructions from the dark thread and the single instruction from the light thread can issue. In the second cycle, both threads stall, and in the third cycle, the dark thread issues its last instruction and the light thread its three remaining instructions. Due to the flexibility of a wide-issue core both threads can run faster in this example. However, in other cases they may also compete for resources.

Simultaneous multithreading was pioneered by Tullsen et al. [TEL95]. Inside each SMT core, the threads share execution units, branch prediction tables, the physical register file, and L1 caches.[1] The PC, rename logic and Load/Store queues are separate for each thread. SMT has been implemented under the name Hyperthreading in the Intel Netburst architecture used by the Pentium 4 and Xeon processors [KM03].

With new designs aggressively pursuing thread-level parallelism, combining multiple threads per core and multiple cores per chip is not a big leap. Such combinations are already implemented in the IBM POWER5 chip [KST04] and the Sun UltraSparc T1 chip [KAO05].

7.2 Loop-Level Threads

Some applications have a limited amount of module-level parallelism. These applications may still have useful parallelism if other sources of speculative threads are used. The survey of TLS architectures in Chapter 2 showed that loop-level threads have been used extensively in thread-level speculation research. Therefore, the new simulation model is extended to make use of loop-level threads in addition to module-level threads.

In loop-level speculation new threads are spawned when execution reaches a loop. Successive loop iterations are run in parallel instead of sequentially. In contrast with to module-level parallelism, where one thread at a time is spawned when a call instruction is found, with loop speculation many thread can potentially be spawned at the same time for successive iterations. Loop-level parallelism can also be exploited efficiently without support for out-of-order spawn. However, if it is desirable to be able to spawn threads for multiple levels of nested loops, out-of-order spawn is still necessary.

A simple example of loop-level speculation is shown in Figure 7.2. It is a for-loop running for five iterations, and has an induction variable *i*. The code is shown to the left in the figure, and in the sequential thread running the program in in the center. *Start* and *end* show the first and last instructions in the loop, respectively. The *bb* labels show backwards branches at the end of a loop iteration, where the control is

[1] This configuration is not mandatory for an SMT but a configuration proposed and commonly used in SMT research to achieve low overhead for SMT support. One could, for instance, use separate register files and branch prediction tables for performance reasons.

transferred back to the top of the loop unless it is the last iteration. The backwards branches are important, since they are used to identify loops.

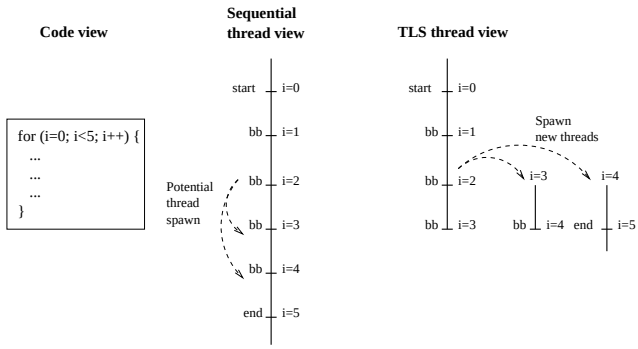


Figure 7.2: *Loop TLS example: Thread spawn for a simple loop.*

When a previously unseen backwards branch is detected, the branch and branch target addresses are recorded. If execution proceeds in the same loop body again, i.e. does not jump outside the address range delimited by those two addresses, and the backwards branch is found again, execution is likely to be in a loop. At this point, the speculation system may choose to spawn new threads for successive iterations. The initial program counter (PC) for the new thread or threads is set to the branch target address and the end of the current thread is the address of the next backwards branch.

Loop thread spawn is illustrated in the thread view to the right in Figure 7.2. The need to find a backwards branch twice before a loop is detected means the first two iterations are lost and can not be parallelized. At the second backwards branch, however, two new threads may be spawned, starting at the fourth and fifth iteration respectively. These threads will run in parallel with the parent thread executing the third iteration, as illustrated in the TLS thread view. Note that the code after the loop, the *loop continuation*, will be executed by the most speculative child thread after the last loop iteration.

In the example, knowledge of the number of loop iterations is assumed. In a real implementation, this is not necessarily available. Spawning threads for future loop iterations speculatively could cause a control misspeculation if it turns out a thread is spawned when there are no more iterations left, or if the loop detection mechanism failed and there was no loop after all. The parent thread needs to be monitored and if it leaves the hypothesized loop body or reaches the last instruction with another next PC than the one assumed as start PC for the child thread, the child thread will be the victim of a control misspeculation. If that happens, the child thread and any subsequent threads must be squashed. Due to limitations in the simulator used in this

work, however, control misspeculations are not modeled. No threads will be spawned along a wrong thread of control.

While it would be possible to spawn new threads after the first time a backwards branch is detected, there are two reasons for waiting until the second backwards branch. Firstly, to avoid starting threads for one-shot backwards jumps and thereby reduce the number of control misspeculations. Secondly, it opens up the possibility for some profiling and prediction ahead of thread creation. Typically, it is important to be able to identify and predict loop induction variables. This is described in Section 7.3.4 along with run-length prediction for loops, which also makes use of this delay.

Figure 7.3 shows a more complex example of loop-level parallelism. In this example, the main function contains two loops, the *k*-loop is nested within the *i*-loop. Both loops consist of four iterations.

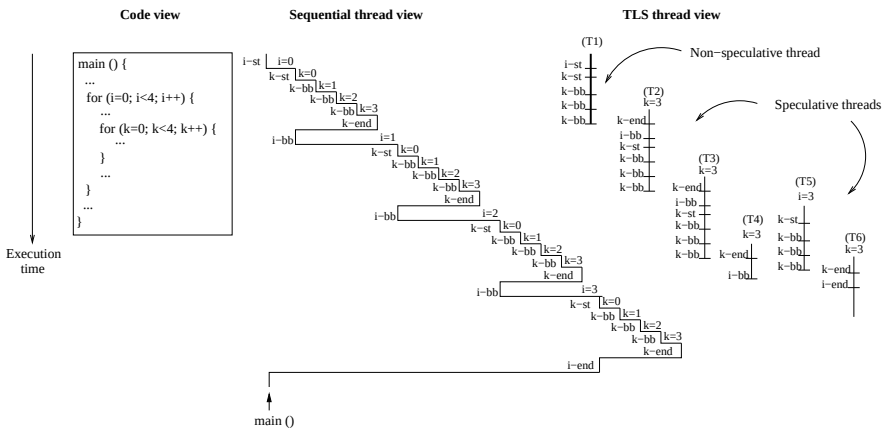


Figure 7.3: *Loop TLS example: Code snippet with two nested for-loops, sequential execution compared to loop-level speculative threads.*

The sequential thread view in this example is shown differently in an attempt to better illustrate the nested loops. The beginning of a new loop iteration is shown as a jump to the right for the thread, while jumps to the left show the backwards branch at the end of the loop.

The TLS thread view in Figure 7.3 illustrates how the speculation system may choose to exploit the parallelism in the nested loops. The non-speculative thread *T1* will detect the first backwards branch *k-bb* and record this in the loop table. Later, a second *k-bb* is encountered and now a new thread *T2* is spawned for the last iteration of k, running in parallel with the second iteration. After the first k-loop has ended, in *T2*, the first backwards branch for the i-loop is detected. Thread *T3* is spawned

in a similar manner. However, within *T3* the second *i-bb* is found, and thread *T5* is spawned for the fourth iteration of the *i*-loop. When *T3* later finds the second *k-bb*, the new thread *T4* is spawned. For this to be possible, out-of-order spawn is necessary, as *T4* is less speculative than *T5*. Finally, *T6* is spawned for the last *k*-loop.

This example illustrates that with nested loops, and loops with few iterations, this thread spawn mechanism may lead to some load imbalance. It is not as accurate or regular as compiler-based loop speculation schemes that also often speculate only on one loop level at a time. However, like the other techniques explored in this thesis, *it will work on unmodified sequential binaries*.

There may not be as many processors as loop iterations. In fact, this should be common. If that happens, the most speculative loop thread will continue to execute the loop after its assigned iteration. Should a processor become available at any point during the execution of the loop, this most speculative loop thread may resume speculation by spawning new threads for the still active loop.

This scheme is adapted from the dynamic loop detection technique presented by Tubella and González [TG98]. They also propose and evaluate data structures for storing information about loops and specific loop executions, including structures which can be used for various types of loop prediction information. For instance, the number of control misspeculations can be reduced by recording the number of executed loop iterations and using this information if the same loop is encountered again. This has not been implemented and evaluated in my simulator, since I do not evaluate the effect of control misspeculations. However, it is likely to be useful in a real system. Such prediction information could be integrated with the other prediction techniques presented in the previous chapters, since they also store information for potential thread spawn points.

7.3 Building Blocks of a TLS Architecture

This section will present the basic mechanisms for a TLS architecture. In this flexible architecture, there may be one or more processors, with one or more threads running on each processor. This may be a chip multiprocessor, an SMT processor, or a hybrid of both.

The target machine is illustrated in Figure 7.4. It has a number of processor cores, single- or multiple-issue. Each core has private level one data and instruction caches, and there is a shared large level two cache. The cores and L2 cache are connected by an on-chip bus. Speculative state can be stored in all data caches, including the level two cache.

This machine model is not mandatory for the TLS implementation. For instance, it would work without a shared L2 cache. However, this is a typical CMP design. The versioning protocol described in this section relies on the ability to broadcast

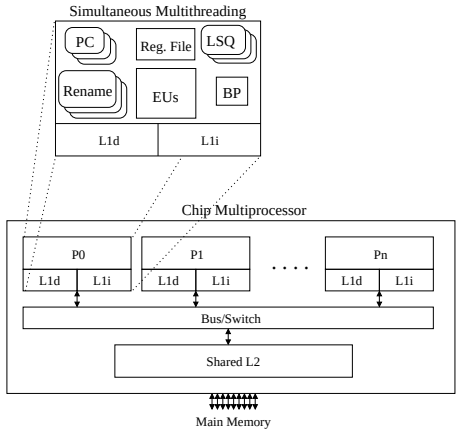


Figure 7.4: *Multithreaded processor: SMT cores in a chip multiprocessor configuration.*

speculative writes and read requests. Therefore, a shared bus with a snoopy cache is used. For a CMP with a different interconnection network, modifications to the versioning protocol would be necessary.

The description of the implementation is divided into the same components as the description of a speculation system in Chapter 2: thread selection and thread-start, how to manage speculative state, and how to commit or squash threads.

7.3.1 Thread Selection and Thread-Start

Starting a new thread requires assigning a free hardware thread and a thread identification number to the new thread, as well as starting up execution on that processor by transferring the required initial values (typically a number of registers, and the start PC) from the spawning, or parent, thread. When it comes to the register transfer, there are a number of ways to implement this. All registers could be transferred, or one could annotate the input registers needed for each possible thread spawn point. Annotation requires compiler support or binary translation. In addition, the registers could either be transferred all at once, before the thread is started, or on-demand transfers could be implemented. In the baseline model, transfer of all registers as part of the thread initialization is pessimistically assumed. This requires more transferred registers than will be used, but is the least complex option. Thread identification numbering is discussed in Section 7.3.2.

For thread-start on a remote core, the registers are transferred via the level one caches and the on-chip communication network, i.e. no dedicated register bus is used. The registers are written to regular memory locations which are then read by

the remote core. It is assumed the registers can be nicely packed onto a few cache lines.

Thread-starts within one core, that is when the parent and the new thread, or child thread, execute on an SMT, is assumed to be a fast operation. With some extra hardware support beyond a normal SMT, the registers could be copied over to the new thread within the core. In fact, depending on implementation details maybe only the register map and PC need to be duplicated [AD98, PV03]. I will assume that register copy within the core can be overlapped with other operations necessary for thread-start (assigning a TID, fetch the first instructions for the new thread etc) so the total time for thread-start is lower than when the registers need to be transferred to another core through memory.

While thread-start is not implemented in detail, i.e. the register flush and read scheme is not actually fully implemented, the simulator supports adding extra thread-start, re-start and commit cycles as well as bus traffic. This scheme is used to approximate the overhead of these events. Except for register transfer, thread-start overhead could include time for checking prediction tables or applying scheduling policies.

As a baseline, a new thread can not be started if there is not a free processor available. Processor affinity is exploited – if a thread has been executed on a processor once, that processor will be the first choice if executing the same code again. This in order to improve cache locality and branch predictor performance.

Module-level threads are, as described before, spawned at call instructions and terminated at return instructions, while the dynamic loop detection technique described in Section 7.2 is used to identify suitable spawn points for loops.

7.3.2 Memory Hierarchy and Speculative State

Buffering of speculative state is done in the cache hierarchy, both in the private L1 caches and the shared L2. In a regular shared-memory machine, there is only one version of any given memory address, though there can be many copies of that version. A coherence protocol makes sure that all processors have the same view of what the memory contains. For instance, as one processor modifies the contents of a memory location, the changes are propagated by some mechanism so that the other processors will get the new version upon reading the same location later at a later point. When using the cache hierarchy as a buffer for speculative values, this scheme needs to be extended. Several schemes have been proposed for buffering speculative versions of memory in the cache hierarchy by extending the coherence protocol. The scheme described here is most similar to the schemes of Steffan et al. [SCM97, SM98] in the STAMPede project, and by Renau el al. [RTL+05, RSC+05]. There are, however, several other related proposals [CMT00, CW99a, GVSS98, OKP+01].

The desired properties for storing speculative state are discussed in Chapter 2. As

a short re-cap, each speculative thread should be able to have its own version of any memory location, and that version must be stored separately from the other versions. Furthermore, a load from a thread should read the thread's own copy of the location if available; otherwise, the value from the closest less speculative thread having a copy should be used. That is, the most recent previous definition of the value in program order should be read. A store, on the other hand, should modify the value in the current and all more speculative threads, i.e. threads that are after the storing thread in program order. A dependence violation occurs when a store modifies a memory location which has already been used by a more speculative thread.

The system should also support out-of-order spawn and allow for several threads to store their speculative state in the same cache. The former is necessary for module-level parallelism and the latter for SMT processors with a shared level one cache. It is also desirable to avoid bottlenecks in the form of burst commits and centralized structures. The chosen system fulfills these goals at the expense of some amount of storage space overhead and version comparison logic.

I will begin by defining a thread ordering mechanism and then continue to discuss the necessary changes to the caches.

Thread Ordering

The importance of supporting out-of-order spawn has been stressed before. The scheme I have chosen to keep track of the sequential order of all threads, is to assign a unique number to each new thread – the thread identification number or *TID* for short. Conceptually, the TID reflects the original sequential order. A thread with a certain TID is located later in the sequential order than all threads with a lower TID, and earlier than all threads with a higher TID. In practice, since there is a finite number of available TIDs, determined by the number of bits used to store the number, the TIDs are recycled continuously. Consequently, TID numbers sometimes wrap around so that a lower TID number is later in the execution than a higher. This needs to be taken into account when determining which thread is later in the sequential order.[2]

Figure 7.5 shows examples of thread-spawn and TIDs assigned to the threads. For instance, thread *T1* has TID 1 and *T2* has TID 50. Since *T2* is spawned from *T1*, thread *T1* is called the parent of *T2* and conversely *T2* is the child of *T1*. A thread's ancestors are all threads it depends on. For *T3* the ancestors are *T1* and *T2*. If any of the ancestors are squashed, the child thread will also be squashed since it may have been affected by the error introduced by the misspeculation.

Note that for module-level threads, the call is illustrated with a jump to the right

[2] In practice, the speculation system keeps track of the highest and lowest TID currently in use. If TID-high is smaller than TID-low the sequence has wrapped around, and the TIDs below TID-high are really sequentially after those larger than TID-low. This knowledge is used when comparing TIDs.

in the figure, while the new thread begins at the module continuation. Therefore, a more speculative thread is shown to the left of a less speculative thread. For loops, I find it more intuitive to show more speculative threads to the right, even if this means an inconsistency in how the speculative order is shown.

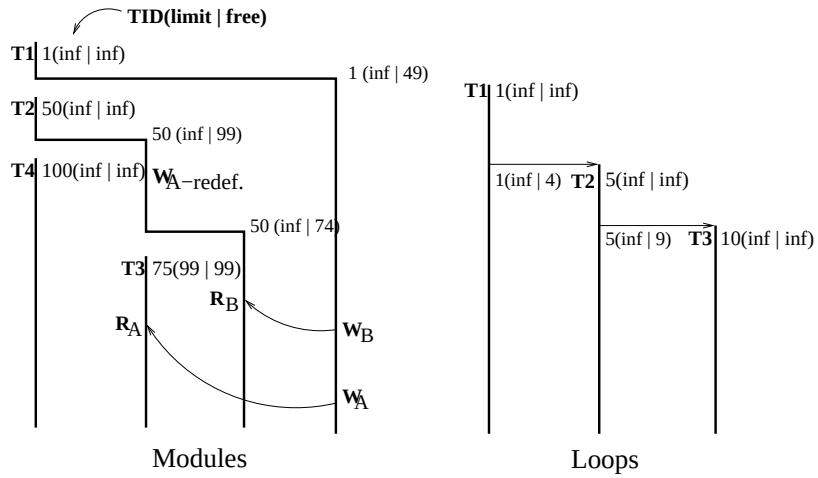


Figure 7.5: *Example showing thread spawn for modules and loops.*

To support out-of-order spawn a gap is left between assigned TIDs. With this strategy, out-of-order threads can later be assigned a TID in between two existing threads and thereby get the right sequential order with respect to the existing threads. This basic scheme was first proposed by Steffan et al. [SCM97]. If there are no TIDs left between two threads, it is no longer possible to spawn new threads with a sequential order in between those two. The most flexible solution would be to keep a dynamic list of the thread order and thus be able to start new threads any time without having to worry about gaps and finding a free TID. This is done in some TLS proposals [HWO98, AD98]. However, this creates an overhead of maintaining a dynamic thread order list, and makes it difficult to construct a speculative state store that enables local dependence checking, flexible thread scheduling and migration.

In my version of the scheme, three different TID values are associated with each thread. The most important is the thread's assigned TID, which defines its place in the sequential order. The other TIDs are used when spawning, committing, or squashing threads. The *free* value keeps track of the gap available between the current and the next used TID, i.e. it points to the last unused TID before the next assigned one. When spawning a new thread any TID in the gap between the parent TID and *free* can be used for the child thread. I define a standard gap for module threads and one for loop threads. If there is space available, the new thread is assigned the TID of

the parent plus the standard gap. If the gap between TID and *free* is smaller than the standard gap, the new TID will be assigned half-way between the parent's TID and *free* value.

The *limit* value defines how far a violation affects the successor threads. That is, the span contains all threads that have the current thread as an ancestor. If the thread has to roll back, all threads within the limit must also be rolled back. Finally, the *next* thread is sometimes mentioned. *Next* is the thread after the current thread in program order. Since free points at the last unused TID, next is simply *free*+1.

On the right side in Figure 7.6, a structure called the TID list is shown. The TID list is indexed with the TID and for each thread contains, among other things, the *free* and *limit* values. The TID list also has a bit which keeps track of the current non-speculative, or head thread. The remaining fields will be explained later. A full TID list will be very large for a reasonably sized TID. This would preclude the list from being stored in a hardware structure close to the L1 cache. One possibility would be to keep the list in memory, and have a small TID cache. This should be feasible since the number of active threads will be much smaller than the total number of available TIDs. Another possibility is to keep the *limit* and *free* parameters in memory since they are accessed less frequently than the other information, only when spawning, squashing, or committing threads.

The speculation system could be extended with the ability to have several applications running in TLS mode simultaneously. In order to facilitate that, the TID can be extended with a task identifier so that the speculation system can keep threads with the same TID but from different programs separate.

Hydra [HWO98] can support out-of-order spawn. The scheme described here, as opposed to Hydra, does not use a special speculation buffer for the speculative state of each thread. This makes it easier to support speculative state for many threads at a time. This is especially important when it is necessary for performance reasons to be able to keep more thread contexts than active threads (see Section 3.4.4), and makes the system more scalable. In addition, no centralized speculation unit is necessary to assign and keep track of thread order. STAMPede [SCM97] uses a similar scheme to assign thread identification numbers. However, they keep thread information in a hardware structure that is tied to the level one cache, which means the thread is pinned to one processor. The scheme presented here is very similar to the one proposed by Renau et al. [RSC+05].

Cache Extensions

The caches are modified to keep track of the speculative state. At the bottom of Figure 7.6, the different fields of a cache line are shown. The bits added for speculation support are *exposed load* and *store* bits.

In my implementation, there is one exposed load and one store bit per word in the cache line. This is to avoid triggering dependence violations due to false sharing. The number of bits per cache line is a trade-off between cache overhead and the amount of false squashes. However, this issue has not been investigated further.

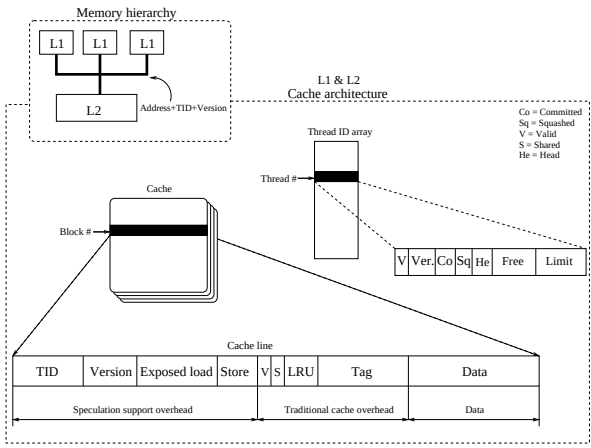


Figure 7.6: *Memory hierarchy with speculation support. Note that several optimizations can be added to this baseline scheme.*

In addition, the TID of the thread the line belongs to must be stored in the cache line since speculative state from many threads can coexist in the same cache. The TID works essentially as an extension to the cache tag, but is only used for speculative threads. The *version* field is a part of the TID - its function will be discussed in the Section 7.3.3.

The dependence detection scheme works similar to the SVC scheme described in Section 2.2.3, except that the TIDs are also used. When a thread writes to a location, the TID of the writing thread is compared to the TIDs associated with all other versions of that location. If the TID of the writing thread is lower than the TID for another version, the other version belongs to a more speculative thread. If the more speculative thread has the exposed load bit set for the location being written, a dependence violation has most likely occured[3] and a roll-back will be triggered. In the SVC, the version was tied to the cache, while the version is explicitly stored in the cache line with this scheme.

[3]Likely and not definitely, since there are some cases where this detection mechanism could detect a violation where none would actually have occurred, but this depends on the dependence detection implementation. Examples of when a dependence could be falsely detected are: silent stores, false sharing, and when a parent has redefined a location. In my implementation, silent stores are eliminated, false sharing could happen on sub-word accesses, and the last category may cause false violations.

Different versions of a memory location are stored in the different ways of an associative cache. Since they will have different TIDs but the same tag and index, this makes finding all available versions of a location as fast as looking up the location in the caches. This also means the upper limit of the number of versions of a single location that can exist in the system is the number of ways in all L1 caches + the number of ways in the L2. If high associativity in the L1 cache is difficult to implement, an alternative may be a small victim cache [SCM97].

The major difference from protocols where the TID is not stored for each cache line is the fact that this scheme allows for the speculative state of several threads to coexist in a cache. This is necessary to allow for TLS on SMT processors where the threads share the L1 cache. This is not the only advantage though. This scheme means that speculative lines can be evicted to the L2 cache, and that threads can migrate to another core without being squashed. As long as the TID is copied with the cache block, the block is not tied to a specific cache. However, the possibility for thread migration is not used in the simulations in the following chapters.

Unfortunately, storing TIDs with every cache line comes at the expense of higher cache overhead. Ideally, the TID should be a reasonably large number. Since TIDs are assigned with a gap to allow for out-of-order spawn, there will necessarily be many unused numbers in the sequence. If the total sequence is too small, it will be difficult to assign TIDs without running out of numbers in some gaps. In the simulations a 32-bit tid is used. Together with the 8-bit version, exposed load, and store fields, this adds up to 56 bits. With a cache line size of 32 bytes, the overhead is 22%. Renau et al. [RSC+05] describe a clever optimization for reducing this overhead. It is possible to let each core maintain a shorter list of local TIDs together with translation from and to a global TID. The global TID is used in version comparisons and for communication with other cores and is stored in the local TID list. The local TID is used for cache-tagging and look-up in the local TID list. Again, this works since the number of active TIDs should be much lower than the TID range. This technique reduces both the size of the TID list and the overhead in each cache line. In addition, using a smaller number of bits for the version field should not be a problem.

Reads that do not hit in the local L1 and writes to shared cache lines from speculative threads need to be broadcast. For writes this allows dependences to be detected in other caches, and data can be forwarded to more speculative threads that have not yet used it but have already loaded the cache line. The write traffic could be reduced by tagging cache lines where multiple versions exist; that way, writes to blocks where only one version exist would not have to be broadcast. For reads the broadcast is needed to obtain the correct version of the data. The version may be in any of the L1 caches or in the L2, so a local TID comparison does not suffice to determine which cache should supply the data. Providing an efficient implementation for reads is an open question. Renau et al. [RSC+05] have approached this with a ring bus

where read requests are passed along to all L1 caches and a victim cache in turn. For a CMP with many processors, a more scalable solution would be useful. The SVC [GVSS98] uses special centralized version control logic to make sure the right version is obtained; this model is also used in my TLS simulator.

Reusing Clean Data

Care is taken to be able to reuse data when possible. For instance, when a block is loaded from memory for a speculative thread, a clean, or non-speculative, copy is kept in the L2 cache. Therefore, if the thread must restart and the cache block in L1 has been modified, the data is not further away than the L2. Furthermore, when a clean block in an L1 cache is speculatively modified, the block is copied if there is a non-speculative cache way free for that tag. That is, a copy-on-write scheme is implemented. These policies aim to reduce the memory latencies, especially for restarted threads.

Register-Carried Dependences

In addition to memory-bound dependences, there can be register-carried dependences. That is register-allocated global variables, return values, or loop-carried dependences may cause dependence violations for module- or loop-level threads. The speculation system must detect and correct such dependences as well. In my system, this works similarly to the system used by Oplinger and Lam [OL02]. When a thread has finished, its final register values are compared to the input registers used by the successor thread, i.e. registers with exposed reads. If the values differ, there is a violation and the thread executing the continuation or next iteration must restart. In order to facilitate this, the initial register values for each thread are kept until the parent thread has committed and performed this register dependence check.

Stack Data Dependences

As discussed by Steffan et al. [SCM97] there will be unnecessary violations through the stack for module-level speculation. This is because the stack space is continually reused. Two functions executing after each other will use the same stack space since the first function resets the stack pointer before returning. When run in parallel, there may be violations between the threads for variables that are in fact local to the thread. Therefore, a module-level thread should operate on its own private stack.

The STAMPede project [SCM97] uses a pool of small stacks, called stacklets, is used. Every new thread is assigned its own stacklet. Stacklets are reused after the thread has committed. In this work, threads keep track of the initial stack pointer of the current function instead. Stores from less speculative threads to the current stack

space are filtered out, since they would not cause violations or update the local stack variable in the sequential case.

7.3.3 Commit and Squash

When squashing or committing threads the speculative values in the cache need to be invalidated. When committing a thread that has finished successfully, its modified data should be merged with the regular, or non-speculative memory state; the system can only have one non-speculative version of each memory location.

Since speculative state from several threads may be mixed in a cache, the technique used in the SVC from Section 2.2.3 will not work, i.e. gang commit or invalidate all speculative values in the cache. A brute-force approach would be to sweep the cache and write back all modified lines found for the committing thread to the lower levels of the memory system. Or, for a squash, invalidate all lines that belong to the thread. However, this would be very inefficient, as there would be a torrent of writes at each commit or squash, which would slow down other running threads and tie up the processor for some time.

Another scheme is used in this architecture. For a commit, the commit bit for the thread is set in the TID-list, and the next thread (found by using *free*+1 in the current thread) becomes the new non-speculative thread. Similarly, to squash a thread the squash bit is set. The squash and commit bits have to be examined for all data accesses to speculative values; if a line that is accessed is found to belong to a squashed thread, the line is invalidated.

However, the data from the committed threads needs to be consolidated at some point in order to free up the TID. Consolidation means making sure only the most recent committed version of each location is kept. For squashed threads, all blocks must be invalidated before the TID can be reused. Consolidation and invalidation can be done on-access, that is when data from a committed cache line is requested. However, it cannot guaranteed that this will clean all cache lines that contain speculative values. Some lines may not be accessed again. This means that walking the cache in search of old speculative blocks lines is still required. The upshot, however, is that this can occur far less often than doing it for every squash or commit operation. If the cache sweep operation is initiated some time before all TIDs are used up, the operation is off the critical path and can be done in the background during spare cache cycles. For simplicity, I assume this is always possible in the simulations, and add no time for cache sweeps.

A similar scheme is used by Cintra et al. [CMT00]. Renau et al. [RSC+05] propose another method to manage cache sweeps in a more energy-efficient way. The STAMPede project [SCM97] uses a list of cache blocks belonging to each thread held in a hardware structure. At commit each block in the list is marked non-speculative,

for a squash all blocks are invalidated. However, this may lead to bursts of traffic. In addition, the need for a relatively large hardware structure for each thread restricts the number of thread contexts that can be supported.

One remaining question is how to handle squashes when the thread will be restarted immediately, which is often the case. One cannot simply restart the thread with the same TID, since there will be dirty cache lines with that TID in the cache. These cache lines may not be reused. Scrubbing the cache for squashed data before restarting the thread is, again, time-consuming. Another option is to just leave all values behind, set the squash-bit to show that all values with this TID are invalid, and assign a new TID for the thread when it is restarted. This has two other disadvantages: If there is no longer a TID available that is in the correct program order compared to other threads it cannot be restarted, and non-dirty values in the cache from the failed execution can not be reused without probing the other caches for more recent values.

To solve both problems, an additional value is assigned to each TID; a *version* number which is incremented when the thread is restarted. Functionally, the version number is not different from reserving the low bits of the TID for restarts, except there may not be another thread using a number in between versions. Therefore, a restart includes increasing the version number. If the restarted thread finds a clean cache block with the same TID but a lower version, the block can be immediately promoted to the new version number without accessing remote caches. The version number does not have to be large, since it is seldom useful to restart a thread many times. Should the thread run out of versions, it is stalled until it becomes the head thread.

Commit overhead includes checking for register dependence violations and possibly updating prediction tables.

Squashing Due to Lack of TIDs

If there are no free TIDs between two threads and a new thread would like to start in that space, we can allow the new thread to squash everything after the parent TID in order to free new TIDs for itself and possible children. The rationale behind this is that we can inadvertently throttle the supply of new threads without this feature. For instance, an old thread very far in the future compared to the head may remain in the system for a very long time. When the TID gap has been used up, the old thread will prevent new threads from being spawned. In the worst case, the bulk of the program belongs to a thread that can no longer spawn child threads. Thus the potential exploitation of parallelism is severely limited. Experiments show that the performance is improved with this feature.

There are some differences for this type of squash compared to restarts. First, the thread is not restarted. Second, a cache sweep must be initiated and completed

before the TIDs can be reused. Third, the limit and free values of the most speculative remaining thread must be adjusted to the new range of unused TIDs.

Squashing Due to Lack of Cache Blocks

Speculative state can not be written back from the cache hierarchy before the thread is committed. In some cases, the caches may not be able to contain all speculative state. Specifically, the cache may not have a high enough degree of associativity to hold all needed versions of a specific address. If this happens, the most speculative thread holding a block with the desired tag is squashed to make sure less speculative threads can continue to execute. When the thread is squashed, its cache blocks are released and may be reused by other threads. This squash policy works exactly like squashing due to lack of TIDs, the only difference is the triggering condition.

7.3.4 Prediction Techniques

This section covers the implementation of previously used prediction techniques, and in particular how they are extended to loop-level threads.

Return Value Prediction

A technique that has proved to work relatively well for module-level speculation while being easy to implement is simple return-value prediction. Stride value prediction was investigated in Chapter 3, and is reused in this new simulation model.

In order to implement return value prediction, a prediction table is needed. For stride value prediction, the return value of the instance of a function that finished execution last is recorded, as is the difference between the last two return values. When the function is called again, the last value plus the stride is used as a prediction for what the new invocation will produce. While the prediction table was infinite in Chapter 3, it is now of finite size and is indexed with the target address of the call instruction. The table is illustrated in Figure 7.7. The table is updated if the predicted value turns out to be wrong.

The prediction table can either be stored in memory or a dedicated local or global hardware structure. A global table accessible by all processors is preferable for faster warm-up, but may have scalability problems. The table is accessed when a new module-thread is started or finished.

Next-Iteration Register Value Prediction

Similar to return value prediction for modules, value prediction is used to predict loop-carried dependences for loops. Since loops typically include induction vari-

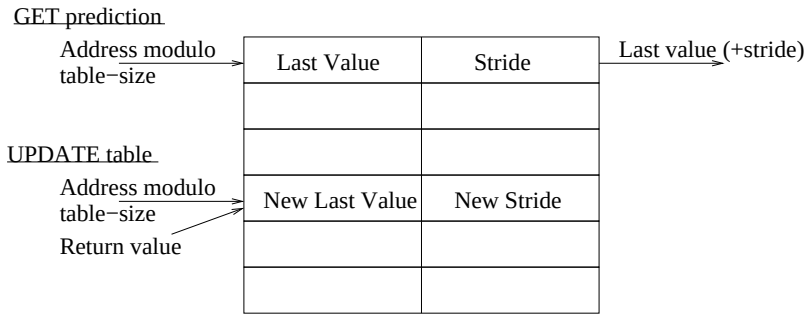


Figure 7.7: *Return value prediction table.*

ables, stride value prediction is suitable for removing these simple dependences. The mechanism is similar to stride prediction for return values. The two main differences are that it can be useful to have more than one prediction for each entry, i.e. each loop, since there may be several loop-carried dependences. In addition, the register containing the dependence is not fixed; thus, the register number being predicted must be recorded together with the prediction. Other dependences, like reduction variables or loop-carried dependences with a more complex update pattern than can be captured with the stride predictor, will cause a dependence violation.

When a backwards branch is detected, an entry in the loop prediction table is reserved. The initial contents of all registers are stored. When the same backwards branch reappears, the registers are compared to the saved values, and the stride predictor is initialized. An entry in the prediction table is shown in Figure 7.8. In this way, induction variables are dynamically identified and predicted. The predictor is now ready to be used to spawn additional threads for future loop iterations.

Loop ID	Reg	Value	Stride	Reg	Value	Stride	Reg	Value	Stride	Reg	Value	Stride	Reg	Value	Stride	Active

Figure 7.8: *Next iteration register value prediction table.*

Since this required information needs to be stored in a loop prediction table, the number of predicted registers need to be limited in order to save space. This should not present a huge problem, since prediction accuracy is likely to be low if there are many loop-carried dependences, so supporting a large number of predictions per loop will not be useful in any case. For the simulations, a maximum of four predictions per loop are supported.

For efficiency, I have used a direct-mapped loop prediction table. This also means there may be conflicts. In contrast with the return-value predictor, conflicting threads may not share a predictor. Instead, if an entry is occupied a second loop with the

same index can not use the prediction mechanism. One could use an associative table if the hit rate for a direct-mapped table is low. The table is indexed with bits from the backwards branch address concatenated with bits from the loop ID. The loop ID is the same as the TID for the parent thread. This number can be used to separate loop threads from the same static loop spawned from another parent thread. The active field stores the number of current active speculative threads in this particular loop. The number is decreased when threads commit, and when this number reaches zero, the table entry can be released.

Run-Length Prediction

Run-length prediction was introduced in Chapter 4 and to prevent short threads from being spawned. Short threads are unlikely to contribute to any speedup even if they do not misspeculate since the thread management overhead will eliminate the potential gain from parallel execution. Figure 7.9 (a) shows the module run-length described in Chapter 4.

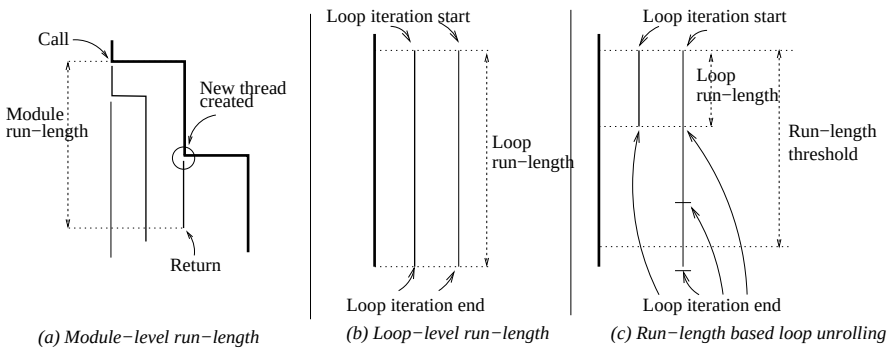


Figure 7.9: *Measuring run-length of a module or loop.*

As the new simulation model introduces loop-level in addition to module-level parallelism, the technique has to be modified slightly to fit the new type of thread. A straight-forward extension would be to measure the length of a loop iteration and simply base the prediction on the length of this single loop iteration. This scenario is shown in Figure 7.9 (c). However, a slightly more advanced variant has been chosen. It has the possibility to unroll loops, i.e. let a loop-thread consist of more than one loop iteration. Instead of recording only whether the run-length exceeds the threshold or not, the number of iterations needed to exceed the threshold is stored.

If the threshold exceeds the measured run-length for a single iteration of the loop, the expression *ceil(threshold/iteration-length)* gives the number of iterations that will produce a thread with a run-length exceeding the threshold. In Figure 7.9 (d), an

example is shown where the run-length for a single iteration is shown together with an unrolled loop. In this example, each thread should consist of three loop iterations in order for the threads to exceed the run-length threshold. The use of next-iteration value prediction has to be modified to account for this as well.

The prediction table is indexed the same way as the return value table discussed above. The address of the backwards branch is used. The table only requires a single bit predictor for each function; for loops, a few bits should be used to support storing the unrolling-factor. The run-length table could be integrated with the return value prediction table for efficiency.

Misspeculation Prediction

The goal of misspeculation prediction is to reduce the overhead caused by an excessive number of misspeculations. The technique is described in Chapter 6. Figure 7.10 (a) shows misspeculation prediction type A and Figure 7.10 (c) misspeculation prediction type B.

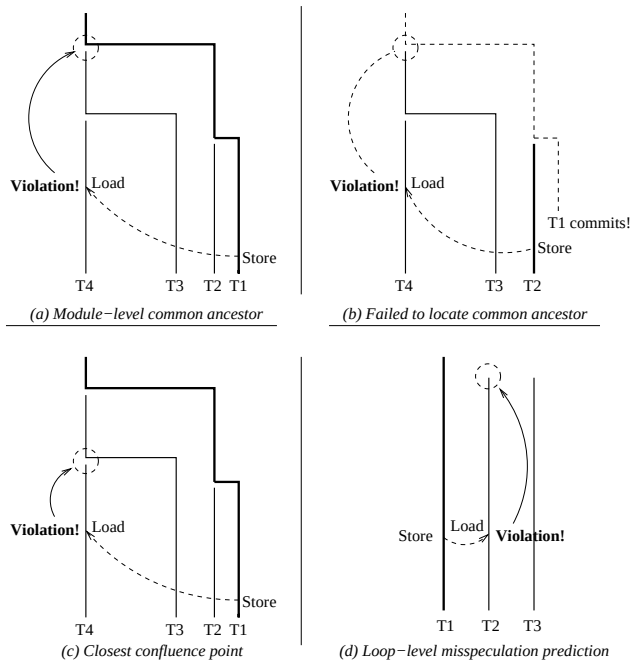


Figure 7.10: *Finding a potential spawn point to mark non-parallel.*

With the new simulation model there is a problem that did not exist with the model

used in Chapter 6. With the TID scheme to keep track of thread order, a thread's information is discarded as soon as it has committed. Therefore, the common ancestor cannot always be found. An example is shown in 7.10 (b) where thread 1 has committed. In this case, the relationship between thread 2 and thread 4 can no longer be determined.[4] Another way to use misspeculation prediction is shown in Figure 7.10 (d). This alternative was called a *type B* predictor in Chapter 6. This variant does not suffer from the problem outlined above, and therefore may be preferrable for this speculation system.

Misspeculation prediction is extended to predict the viability of loop-level speculation. Confluence point identification works similarly to the variant in Figure 7.10 (c), i.e. the misspeculating thread is marked non-speculative. In principle, a scheme like the one in Figure 7.10 (a) could possibly be useful, especially when loop- and module threads are mixed. For violations between loop iterations, however, both variants would yield the same result.

Since, in my model, loops are identified and loop-threads spawned at run-time by analyzing backwards branches, the address of the backwards branch is used to index the prediction table, not the first instruction of the loop. The prediction table looks and works like the run-length prediction table, and the two can be integrated for efficiency. Only one bit per spawn point is needed with the last-outcome predictor or n bits with an n-bit predictor.

7.4 Experimental Framework

This section describes the simulator and framework developed for the detailed TLS model. First the simulation toolchain is presented, and then the sample-based simulation methodology followed by the benchmark applications used in the simulations.

7.4.1 Simulation Toolchain

The simulation methodology with the new TLS model is similar to the methodology described in Chapter 3. First, the source code for the application is compiled for the Sparc platform with the Sun Workshop 6 compiler. The application is then run sequentially on Simics 1.8 [MCE+02], a full-system instruction-set simulator which mimics a SPARC workstation with all necessary devices and runs the Solaris operating system. A trace-generation module attached to Simics captures the executed instructions and creates a trace. The traces are used in the simulations on my trace-based TLS simulator. The toolchain is shown in Figure 7.11.

[4] We know thread 4 is more speculative than thread 2 since it has a higher TID, but cannot find the path between them in order to find their common confluence point.

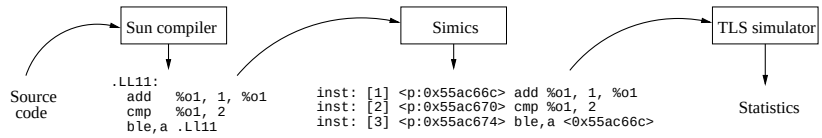


Figure 7.11: *Toolchain for detailed simulation model.*

Trace Generation

The trace module attached to Simics captures all instructions and writes a trace file of the program execution. In order to create a simulation sample, i.e. a trace of some size from a certain point in the benchmark, the program is fast forwarded with only functional simulation to the desired point in the execution. From there, the trace module is activated, first with cache simulation only in order to warm the caches. When the caches are warm, the data is written to a cache snapshot file and the full trace generator activated. As opposed to the previous simulation framework, this trace contains all executed instruction in order to facilitate accurate simulation of an out-of-order processor.

The trace generator also analyzes the instruction stream to find potential thread spawn and completion points. Instructions that may spawn new threads are annotated with some extra information. The location in the trace for the start of the potential thread is recorded, as is the current stack pointer for modules and register content information necessary for register-dependence checks. In order to make use of silent store elimination, the values written by store instructions are also included in the trace. OS overhead is omitted from the trace; the effect of exceptions is discussed below.

TLS Simulator

The trace is fed to my detailed TLS simulator. The primary advantage of using a trace-based simulator instead of running the final simulations on a full-system simulator is the possibility to simulate full thread roll-backs, something which is not easily accomplished in a full-system simulator such as Simics, but necessary for TLS.

The trace engine feeds a superscalar SMT processor core. The processor fetches instructions, does branch prediction, issues read and write memory requests and passes the instructions down the pipeline as a regular out-of-order processor. However, no actual results are produced, since the actual execution is already taken care of by Simics. The instructions fetched by the processors are picked from the trace instead of memory.

There are some simplifications compared to a real system, and some differences that stem from the fact that the TLS simulation is not a full system simulation.

Processors: When there is a branch misprediction, the incorrect instructions the processor wants to fetch are not available, since only the correct execution trace is available in the trace file. Instead, the processor core is fed NOPs until the branch is resolved and the processor resumes execution on the right path. By injecting NOPs, there will be some resources occupied in the processor during the wrong-path execution, though not necessarily the same resources that would have been used during a misspeculation in a real system. In particular, there is no issue of wrong path memory accesses. Another simplification is that the processor has an unlimited number of rename registers.

Memory hierarchy: The on-chip bus is modeled with a fixed minimum latency. On top of this, congestion may add to the total latency imposed by the on-chip bus. If the bus is occupied transactions are queued and completed in a first-come first-served fashion. Main memory accesses, however, are modeled with fixed latency for fetching a cache block. There is no modeling of congestion or other characteristics which could affect the latency for main memory accesses. Furthermore there is no modeling of TLB or page faults.

Regular Exceptions: In general, exceptions have to be executed non-speculatively. However, this depends on the kind of exception. Typically, OS calls must be executed non-speculatively, though it might be possible to make many of these calls TLS safe. For the SPARC processor, there are frequent *register-window exceptions* which are essentially a string of load or store instructions saving or restoring register contents. These could be handled with TLS, but since they are not a significant part of execution (0.01% in a test trace) I simply omit them. *MMU-related exceptions* should not be captured in the trace since they would not occur in the same order in a TLS system. They should also be infrequent after warmup so I do not consider it worth the effort to add a TLB to the simulator. However, with TLS enabled, it is probably beneficial to update the TLBs of all cores in a CMP on a miss, since code sections within the same page will often be spawned to run on different cores. *Context switches* are not modeled since only one running application is considered. In a multithreaded chip, one could easily imagine a number of hardware threads being allocated to a speculative process which is allowed to run for extended periods of time without interruption, while other threads are handled by the OS-scheduler as usual with more frequent preemption. If an application in TLS mode were to be switched out, the best action would probably be to squash all speculative threads first, since the overhead of a context switch is relatively large.

Exceptions caused by speculation: Memory accesses can cause exceptions when executed speculatively. If the address calculation depends on data that is wrong due to an as yet undiscovered dependence violation, the instruction can issue with an illegal address. Since speculative execution should produce the same result as corresponding sequential execution, this exception cannot be allowed to affect the

system further. Instead, the speculation system must stall the thread until the correct address can be calculated. The safe bet is to stall until the thread is non-speculative, but one can also restart the thread and hope next time will be the charm. With my trace-based approach this will never happen; the correct addresses are known a priori (from the trace) and thus no erroneous accesses are issued to the memory system. This will not result in optimistic speedup figures. Even if the would-be exception is not detected (and the speculative execution hence continues) the cause of the illegal address, i.e. the dependence violation, will still occur and be detected. This will cause the offending thread to be restarted in my simulator as well, only the problem will be detected later.

7.4.2 Creating Simulation Samples

It is difficult to simulate an entire program execution with even a relatively small workload. With a large workload like SPEC CINT2000 with reference inputs, it is simply not possible if one want results within a reasonable amount of time. Instead, various kinds of sampling techniques are often used, with the aim of capturing the behavior of the entire program while only simulating small parts of the full execution with a detailed simulation model. There are several important recent works in this area, such as SimPoint [SPHC02] and SMARTS [WWFH03].

SMARTS builds upon statistical random sampling, and works together with a simulator that can fast-forward with simple functional simulation between a moderate number of small samples, which are used to calculate both average performance and error bounds. The samples can be quite small, only 10k instructions in [WWFH03], and yet yield a good result if one is measuring e.g. IPC. A moderate (30+) number of 10k samples can be simulated very quickly. However, for TLS simulations such small samples will not capture the behavior we are looking for, since there are startup and end penalties for each sample. After starting a simulation thread-spawn points must be reached before any parallelism can be exploited and at the end of the sample only threads that have their final instruction within the trace can be started, else one cannot know if it would finish correctly or misspeculate.

Tests with various sample sizes indicate that samples of 25 million instructions are more than adequate to capture the available parallelism without disturbing the results with startup and end sample artifacts. With only four samples per benchmark, the results presented in the following chapters required over 4000 simulations.[5] Each simulation takes from a couple of hours to a couple of days to complete, depending on the complexity of the model. Clearly, using a large number of long samples is not a viable option. Using 30+ samples per benchmark would have severely limited the

[5] Even more, in fact, when counting experiments and verifications which are only briefly mentioned.

design space I could explore. Therefore, I have not been able to use enough samples to compute statistically valid error bounds.

SimPoint cannot guarantee an upper bound on the error compared to full simulation, but has been demonstrated to work well in practice. SimPoint has the advantage that typically only a handful of samples are needed to accurately represent the entire benchmark. The method works by identifying representative parts of the application.

While SimPoints would be a reasonable choice, in this work I opt for a simpler method. Systematic sampling is used, i.e. a number of samples are taken at even intervals during the benchmark execution. The number of samples are, for practical reasons, too small to compute an error bound. Therefore, it should be noted that the results presented may be different for the entire execution. However, the results have been manually inspected, and if variations were found the number of samples has been increased.

7.4.3 Benchmarks

For each benchmark program, simulation points have been created at 20%, 40%, 60% and 80% of the total execution time, i.e. four samples per application, except for *art* which uses eight traces due to a larger variation in the results. The traces are created by running fast functional simulation up until the desired point. Cache simulation is then enabled and the caches are warmed for 100 million instructions, followed by a cache dump. Finally, the trace generator is activated, creating a full instruction trace of the following 25 million instructions. That is, a total of 100 million instructions are simulated for each benchmark, and in all simulation results in the following chapters, the graphs show an average of these four simulation points.

Nine benchmark programs are used. **Art** is an image recognition benchmark that uses a neural network. **Equake** is a scientific application, simulating seismic wave propagation. Art and Equake are from the SPEC CPU 2000 floating point benchmarks (CFP2000). These two are chosen for practical reasons; most of the floating point benchmarks are written in Fortran. I did not have access to a Fortran compiler, nor any knowledge of the language. Art and Equake are two of the four programs written in C.

There has been some success in parallelizing numeric applications with parallelizing compilers. Zhang [ZUR04] attempts to parallelize the CFP applications with an OpenMP based auto-parallelizer. While there were parallelizable loops in art, the results show no improvement for art or Equake. However, the parallelization cost is higher. I have not attempted any compile-time analysis to uncover loops that might be parallelizable even without TLS.

Vpr is an FPGA place and route tool, **Vortex** is an object-oriented database, **Gzip** does file compression, and **Perlbmk** is a cut-down Perl 5 language interpreter. These

programs are part of the SPEC CPU 2000 integer benchmarks (CINT2000). These applications are chosen since they have shown to be interesting, both by my measurements and others' [SCZM02, RTL+05]. Many of the other CINT2000 applications have shown no useful speculative TLP at all.

Finally, **M88ksim**, **Deltablue**, and **Neuralnet** are kept from the previous chapters. M88ksim is from CINT95, Neuralnet from jBYTEmark, and Deltablue is a benchmark from Sun Labs. However, larger input sets than in the previous chapters are used. The previous input sets were small due to limitations in the simulator; it was necessary to include the entire execution in the trace, and some of the simulations, e.g. those with infinite resources, were time-consuming. With the new methodology, it is still too time-consuming to simulate entire runs, but the environment instead supports using samples out of a longer execution, which makes it possible to use larger inputs.

There are two primary reasons why larger input sets are desirable even for the old benchmarks. First, the risk that scaling down the application creates an unrealistic workloads is reduced, Secondly, and perhaps even more important, is that when running simulations with a detailed memory system the results are likely to deviate even more between scaled down and realistic input sets. The small inputs would easily fit in the caches, and scaling down cache size to accurately match scaled-down input sets is a tricky business.

Table 7.1 shows the input sets used by each benchmark. The SPEC 2000 reference input sets generally consist of several parts, i.e. the application is run several times with different input. The second row in the table states which of these inputs are used.

Table 7.1: *The benchmark applications - names (above) and input sets.*

art	dblue	equake	gzip	m88k	nnet	perlbmk	vortex	vpr
ref.110	default	ref	ref.log	ref	default	ref.perf	ref.1	ref.route

8

Impact of Detailed Models on TLS

The experiments in previous chapters have focused on the inherent parallelism in the applications, and the overhead caused by thread-management and roll-backs. A simple single-issue, non-pipelined processor model with ideal one cycle memory access was used to provide a relatively simple model to analyze. With this methodology I have been able to assess the potential of module-level parallelism and identified several performance bottlenecks.

In a real machine, however, there will be additional effects caused by memory access latencies, and latencies when communicating values between threads, especially threads on different processor cores. Most processor cores are also more complex than the one used in the previous experiments. Modern processor cores are pipelined and can generally issue multiple instructions each clock cycle to the many available execution units.

In this chapter, the detailed machine model described in Chapter 7 is used to evaluate the impact of a memory hierarchy in line with that found in modern processors and inter-thread communication overhead, as well as effects of issue-width, out-of-order execution, pipelines, and branch prediction.

Furthermore, the speedup gained from thread-level speculation is classified as resulting from loop-, module-, or memory-level parallelism. To get a complete picture of the potential gain from thread-level parallelism, the trade-off between thread-level and instruction-level parallelism is studied.

8.1 Architectural Models

In the same vein as Chapter 3, this investigation of thread-level parallelism with a detailed machine model will begin with a simple model with relatively few restrictions, and then gradually progress to more complex models. The first three models are loosely comparable to the first models in Chapter 3 but for the new simulation environment. The additional models take advantage of the detailed processor core and communication features of the new simulator.

Model 1: Perfect Value Prediction

The first model uses a chip-multiprocessor TLS machine with the parameters listed in Table 8.1.

Table 8.1: *Baseline machine parameters - single-issue processor.*

Feature	*Baseline parameters*
Fetch/issue/commit width	1/1/1
Execution units per core	1 ALU, 1 Ld/St, 1 fp.
	ALUs are general-purpose integer/branch units.
	Fp are fully pipelined floating point units.
Instruction window size	64
Load-store queue (LSQ)	32 entries.
Branch predictor	G-share,
	16k predictor table, 8-bit global history,
	2k target buffer,
	8 entry Return address stack per thread.
	1 prediction per cycle.
Pipeline length	5 stages integer, 8 fp.
Number of cores	1 or 8.

In this model, each processor in the CMP is a simple single-issue processor. A simplified view of the pipeline is shown to the right in Figure 8.1. There are no overheads for thread-management, memory accesses, or inter-core communication. In addition, perfect value prediction for both registers and memory accesses are assumed. Therefore, there are no dependence violations.

A maximum of 1000 simultaneous thread contexts can be handled by the machine. This number is set sufficiently large not to influence the results. The thread identification number (TID) is a 32-bit value and version is an 8-bit number. For a new module thread, a gap of 2^{20} is left in the TID sequence if available, else the new thread will get a TID halfway between the parent's TID and *free* numbers. For loops, a smaller gap of 2^{6} is used in the same manner.

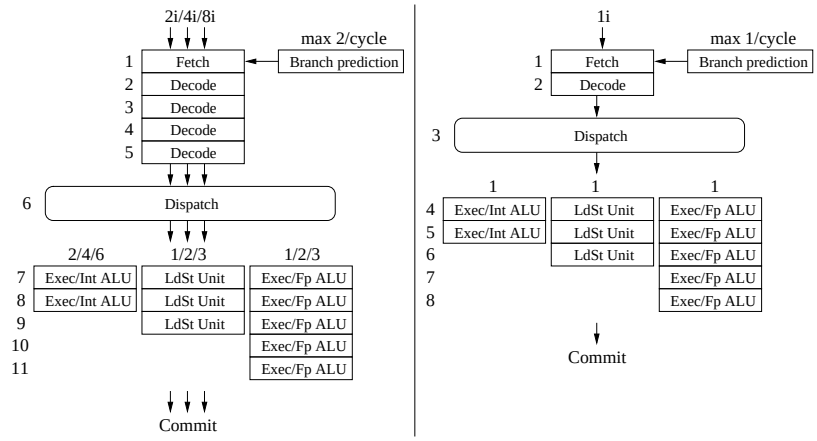


Figure 8.1: *Pipeline for the multiple-issue (left) and single-issue (right) processors.*

There are a number of notable differences from the ideal measurements in Chapter 3. First, it is a pipelined processor core with imperfect branch prediction. In addition, there are not an infinite number of processor cores. The threads are started on a first-encounter basis, i.e. there is no preemption as with the previous simulation models. Finally, even with perfect value prediction there may be some threads squashed due to lack of free TIDs as described in Chapter 7.

Model 2: Return- and Loop Register Value Prediction

While model 1 faked perfect value prediction for all register- and memory dependences, something which is not achievable in the real world, the second model employs feasible value prediction. The stride value predictors for register dependences described in Sections 7.3.4 and Sections 7.3.4 are used; the prediction table has 1024 entries, where each entry can contain predictions for up to three register dependences.

There is no value prediction for memory accesses. However, silent store elimination is used to avoid triggering dependence violations when a store contains the same value as that already present in the memory location. Silent store elimination is, in this case, equivalent to using a last-value predictor for memory locations.

Model 3: Thread-Management Overhead

The third machine model adds thread-management overheads. With this model, fast hardware support for thread-management is assumed; the amount of overhead is smaller than in previous sections where the amount was loosely based on the Hydra

project. Hydra uses a software-based exception mechanism for thread management. The thread-start overhead consists of loading live-in register values from the previous thread, and may include accessing a couple of prediction tables. The mechanisms are described more closely in Section 7.3. In this model the only prediction technique used is register-value prediction. In reality, the overhead should be somewhat larger when adding additional techniques such as run-length or misspeculation prediction. However, in order to make comparisons easier, the same thread-management overheads are assumed for all models.

The amount of overhead is summarized in Table 8.2. Of the listed parameters, only the thread-start, restart, and commit overhead (without bus transactions) apply to this model.

Table 8.2: *Baseline machine parameters - overhead.*

Feature	*Baseline parameters*
L1 caches	32+32 kbytes i+d, 4-way associative
	3-cycle load-to-use latency
	3 extra cycles for version upgrade or block duplication from less speculative thread.
L1 instruction cache	Sequential prefetching
Shared L2 cache	4 Mbytes 32-way associative
	15-cycle total latency without contention
Cache block size	32 bytes
On-chip bus	256 data bits, 2 cycle latency
Main memory	200 cycle latency
Thread-start overhead	20 cycles + 4 bus transactions
Restart overhead	10 cycle + 1 bus transaction
Commit overhead	10 cycles + 4 bus transactions

Model 4: Communication Overhead

This model adds a memory hierarchy and communication network between the processors. The parameters are listed in Table 8.2. The baseline caches are 32 kbytes instruction, and 32 kbytes data L1 caches for each core, and a large 4 Mbytes shared L2 cache.

Four bus transactions are assumed for transferring register values when starting a thread on another core. With the baseline bus width this means there is room for 32 registers, which is the number of integer registers plus the program counter. This is an estimate since thread-starts are not implemented with this level of detail. Four transfers may seem low considering there may also be a need to transfer floating point registers. However, it is possible to reduce the number of transferred registers

by marking the live-ins for potential spawn-points; this could be done in a binary translation pass or even with run-time profiling. Thus, four bus transfers seems like an achievable estimate.

This model also adds a load-to-use latency of 3 cycles for L1 cache accesses, which alone will increase the CPI significantly for several applications. However, it should be noted that in the figures showing memory access latencies, stall cycles due to L1 hits are not included. Miminum latencies are shown, if there is contention for the on-chip bus, memory accesses beyond the L1 cache will take longer.

Model 5: Multiple-Issue Processors

Model five is used to simulate TLS with varying levels of complexity of the cores in a chip multiprocessor with respect to issue-width. The single-issue (1i) core is compared to 2-issue (2i), 4-issue (4i) and 8-issue (8i) out-of-order processor cores. The parameters for each core are listed in Table 8.3.

Table 8.3: *Baseline machine parameters - single vs multiple-issue.*

Feature	*Parameters*			
	1-issue	*2-issue*	*4-issue*	*8-issue*
Fetch/issue/commit width	1/1/1	2/2/2	4/4/4	8/8/8
ALUs per core	1	2	4	8
Load/Store units per core	1	1	2	3
Floating point units per core	1	1	2	3
Instruction window size	62	128	256	384
LSQ size	32	48	64	96
Branch predictions per cycle	1	2	2	2
Pipeline length – integer	5	8	8	8
Pipeline length – floating point	8	11	11	11

The multiple-issue cores have a longer pipeline than the single-issue core to compensate for the more complex front-end logic of wide-issue processors. The difference is illustrated in Figure 8.1. For simplicity, all multiple-issue processors have the same pipeline depth. The number of stages have been chosen by comparing with real designs (e.g. some AMD Athlon, Intel Pentium, and Alpha pipelines). In addition, the wide-issue processors can manage up to two branch predictions per cycle. The fetch and commit widths are the same as the issue width.

With this model, the trade-off between instruction-level and thread-level parallelism can be studied. One potential threat to thread-level parallelism is that the gain from ILP will be reduced when splitting up the program into threads. Multiple-issue processors are typically more deeply pipelined and take a longer time to reach peak efficiency from thread-start as the issue queues have to be filled up with instructions

first. If the speculative CMP consists of multiple-issue processors, high-ILP applications may not see the same performance boost from TLS as they do with single-issue processors, since the gained thread-level parallelism (TLP)[1] will be offset by lost ILP. In addition, the threads will become relatively shorter compared to the TLS overhead with wide-issue cores since the threads execute faster but the thread-management overheads remain the same. Therefore, the overhead may become more noticeable. This model can reveal how big the impact of these effects are.

Model 6: Run-length Prediction

The run-length prediction technique introduced in Chapter 4 is re-evaluated with the new detailed TLS model. This experiment will show how the addition of complex processor cores and communication overhead impact the performance of run-length prediction; it will also show how the dynamic loop unrolling mechanism described in Section 7.3.4 performs.

Model 7: Misspeculation Prediction

Misspeculation prediction was presented in Chapter 6. This model will show how this technique is affected by the overheads in the detailed model. The impact of the problem with identifying confluence points described in Section 7.3.4, and how well misspeculation prediction works together with loop-level threads are other issues of interest.

8.2 Simulation Methodology

The simulations in this chapter are conducted using the methodology, tools, and benchmark applications described in Chapter 7. The processor model as well as the speculation system are highly parametrized, and this flexibility is used to study the performance variation of the different models listed in Section 8.1.

Interpreting the Figures

The contents of the figures will be explained in each of the following sections as new variants are introduced. However, Table 8.4 provides a reference for the abbreviations used in the legends for the figures throughout the chapter.

[1] Throughout this chapter, TLP refers to the speculative thread-level parallelism exploited with my TLS model.

Table 8.4: *Summary of figure legends.*

Legend	*Explanation*
*y*t	Maximum number of running threads (t) (y = 1, 2, 4, or 8).
loops	The keywords *loops* or *mods* show that only loop-level
mods	or module-level threads are used. Default is both kinds enabled.
tmoh	Simulations are with thread-management only (tmoh) or both
oh	thread-management and communication overhead (oh).
	If none of these are specified, the simulation was run without
	overheads, i.e. no thread-start/restart/commit overhead as well as
	all cache, bus and memory latencies set to zero.
*x*i	Issue width (i) of the processors (x = 1, 2, 4, or 8).
rl-*s*	Run-length predictor with threshold *s*.
mpa-*r*b	*r*-bit misspeculation predictor of type A.
	mp-lv (last-value) is used instead of the equivalent mp-1b.
mpb	Misspeculation predictor of type B (only 1-bit pred. is used).
def	Deferred squash is enabled.
def-ra	Deferred squash with runahead mode is enabled.
gmean	Geometric mean.

8.3 Dependences and Overhead

The experiments in this section are similar to the first experiments in Chapter 3. These experiments are run on the detailed simulation model which has some restrictions and overhead the previous model did not have, and the new set of benchmarks are used. In addition, I will look at the performance of a more realistic machine model with communication overhead.

8.3.1 Perfect Value Prediction

Model 1 has perfect value prediction and no thread-management or communication overheads. Results for an 8-way machine are shown in Figure 8.2. In this and all subsequent graphs, the speedup is shown relative to sequential execution on a single processor with the same configurations as the cores in the TLS chip multiprocessor.

Speedups range from two to five with a mean of about three. These numbers are loosely comparable to the results in Figure 3.5 for the old benchmark set. The numbers with the new setup is slightly lower. However, this model does not have unlimited processors, and the results are affected by the influence of pipelined processor cores and imperfections due to TID allocation.

One might have expected speedups with perfect value prediction to be better, especially for applications such as Art and Equake with high loop coverage. The

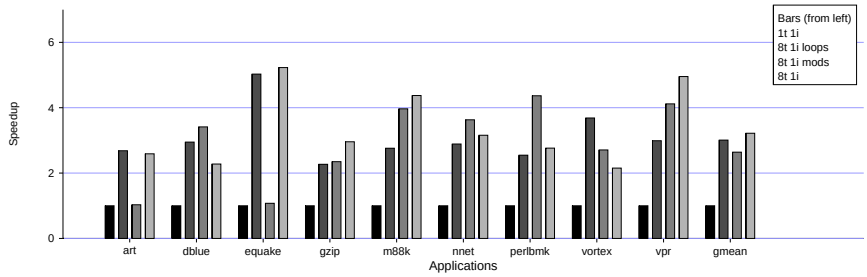


Figure 8.2: *Speedup with perfect value prediction. 8-way machine without overheads, and single-issue processors.*

expectation is fulfilled by Equake, but not by Art.

Art and Equake both have plenty of loops and should be able to approach linear speedup with perfect value prediction. The big difference between the two applications is loop size. The loops in Art are significantly smaller. In Figure 8.3, the threads are categorized according to the number of instructions they contain. Darker shades represent shorter threads. For instance, the black part at the bottom of the bar is the percentage of threads with fewer than 20 instructions. The bar showing loop parallelism for Art is one of the simulations with the largest fraction of short threads. Note that the figure shows the percentage of spawned threads belonging to each category, and *not* the fraction of instructions or fraction of execution time spent executing threads from that category.

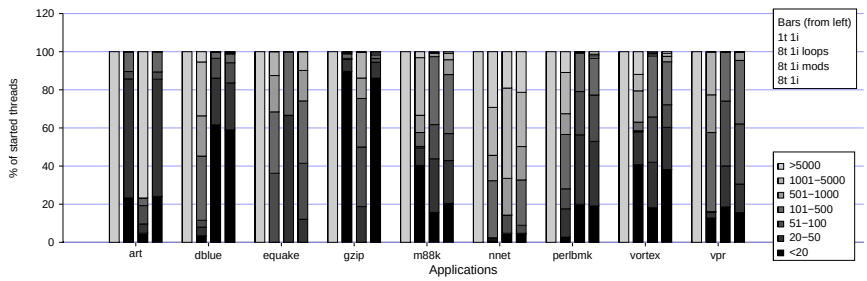


Figure 8.3: *Thread-size breakdown for TLS execution with perfect value prediction.*

Even with perfect value prediction and no memory access latencies, there is some overhead when starting a thread; the processor pipeline needs to be filled before the first instruction can execute, and the last instruction needs to be committed before the processor can be reused. This overhead becomes notable with very short threads. For

Art, over four processors are active on average, but the speedup is below three due to this problem.

An additional reason has to do with the way loop threads are spawned. When a backwards branch is encountered for the second time, threads for future iterations will be spawned for all available processors. The last spawned, or most speculative of these threads, also contains the loop continuation code. No new threads will be started before that thread has again identified the loop in the same manner, and there is at least one empty processor. This leads to some load imbalance every time a new batch of loop threads are to be created. If there are nested loops, the situation is even worse, as shown earlier in Section 7.2 and Figure 7.3. In the case of Art, this factor is significant, and a big contributing factor to why only half of the processors are active on average. There is clearly room for improvement in the loop thread-spawn policy. Improving the spawn policy is, however, beyond the scope of this thesis.

For several benchmarks, the average speedup for loop and module parallelism combined is lower than loops or modules alone. This has to do with the thread identification (TID) numbers. Despite perfect value prediction, some threads are still squashed. This happens when there are no free thread TIDs, i.e. when a thread's *TID* equals its *free* value. When this happens, the speculation system will squash more speculative threads in order to reuse their TIDs. In these simulations a large 32-bit TID number is used. The gap created for module threads is one million, while the gap for loops is 50. The lower number for loops is due to the fact that loop threads are usually created in chunks and there is more seldom a need to create many in-between threads.

When mixing loops and modules, thread spawning becomes a bit more messy, since loop and module threads may be intermingled. In addition, with two sources of parallelism the total number of threads is larger than with only loops or modules alone. This leads to a higher risk of running out of free numbers, which leads to more squashing and lost speedup. In fact, this does happen regularly with this machine model. One might think the problem can be resolved by increasing the gap for loops when mixing both types of parallelism. After testing with various combinations of module and loop TID gaps, I did not find another combination which consistently resulted in better speedups; there are other combinations which perform better in isolated cases though. How to best manage the TIDs is still an open question.

When taking dependences and overhead into account this matter becomes less important relative to other effects. Due to misspeculations, fewer threads are created and squashing due to lack of TIDs is more rare. However, if fewer bits are reserved for TIDs, this problem may remains. *Dynamic task merging* and other techniques proposed by Renau et al. [RTL+05] could help.

For the other benchmarks, the speedups reported are a combination of these limiting factors, and simply a lack of sufficient parallelism. The lack of module-level

parallelism in Art and Equake is expected, as the module coverage for both Art and Equake is very low. As discovered in earlier chapters, module parallelism often does not scale to eight cores. Even if module coverage is high that does not necessarily mean high scalability, just that there are at least two available threads in that part of the application.

To conclude, a slightly lower potential is seen with the detailed simulation model compared to the results in Section 3.4.1. However, the potential is still useful for small-scale CMPs.

8.3.2 Return- and Loop Register Value Prediction

Figure 8.4 shows the results for *Model 2*. Perfect memory- and register value prediction is replaced by stride value prediction for register dependences only, and silent store elimination for memory accesses.

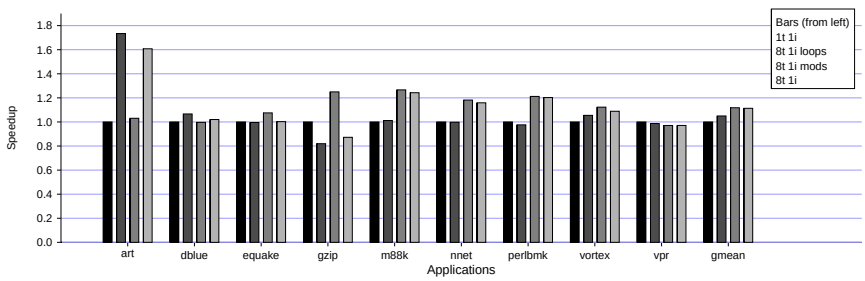


Figure 8.4: *Speedup with register value prediction. 8-way machine without overheads, and single-issue processors.*

In general, the speedups are lower than the comparable model in Chapter 3. Three of the benchmarks from the previous chapters are reused. Of these, the speedups for M88ksim and Neuralnet are notably lower.

The lower amount of parallelism is partly due to the properties of the detailed model discussed above. But to a large extent, the reason is lower parallelism in the larger input sets now used. With the previous simulation methodology, the whole application was simulated with a very small input set. In the setup used here, samples are taken with the default or reference input sets for the benchmarks. For M88ksim, the old setup used a cut-down version of the test input set. A relatively large portion of the execution time for the test input set turns out to be in the initialization phase of the simulated processor of M88ksim, which exhibits large speedups. In the cut-down version, an even larger part was spent in this phase. The remaining execution of the test input set also shows higher speedups than the reference input used now. Together

this accounts for the large difference.

The reason for the lower speedup in Neuralnet is similar. The larger input set limits the speedup gained from overlap of different phases of the execution in the small set. For other applications, like Deltablue, the behavior is similar with the old and new setup.

Gzip shows a slowdown for loop parallelism, which may be surprising when there is no thread-management or communication overheads. However, it is clear from the thread size break-down in Figure 8.3 that the TLS system is attempting to start a large number of very short loop threads. Frequent dependence violations together with the overhead from pipeline startup and flush for thread-starts, restarts and commits are enough to slow down the parallel execution compared to the sequential even without considering other overheads.

Module- and loop parallelism still seems difficult to combine. For Vortex which does have both some module- and loop parallelism, the combination performs worse than module-parallelism alone. There are many more available thread spawn points than processors, and using a first-encountered spawn policy can create problems, for instance with load-balancing and thread sizes. Using both types of parallelism there are even more spawn points and the average thread size is smaller, which can hurt performance. There is likely to be room for improvement in the thread selection policy, especially when it comes to successfully mixing module- and loop threads.

With realistic value prediction, six of the nine benchmarks have a speedup of more than 10% over sequential execution and two show a slowdown when spawning threads for all potential spawn points if a processor is available.

8.3.3 Thread-Management Overhead

When thread-management overhead is added in *model 3*, the speedup predictably goes down somewhat. The results are shown in Figure 8.5.

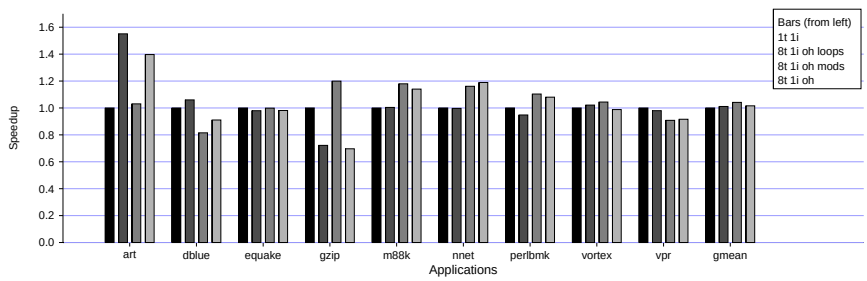


Figure 8.5: *Speedup with thread-management overheads. 8-way machine, and single-issue processors.*

The small speedup due to module-level parallelism in Equake is lost. Vpr and Deltablue show noticeable slowdowns. Overall, the results are as expected. Applications with many short threads take a larger performance hit than applications with longer threads.

8.3.4 Communication Overhead

Impact of communication overhead, according to *model 4*, depends on the characteristics of the application. On one hand, the result for benchmarks like Deltablue, M88ksim, Neuralnet, Perlbmk, and Vortex do not change much when communication overhead is added in Figure 8.6. On the other hand, speedup for Art, Equake, Gzip, and Vpr increases significantly. The reason for this speedup is that TLS not only benefits from parallel execution of actual code, but also from memory-level parallelism. This phenomenon will be studied more closely in the next section.

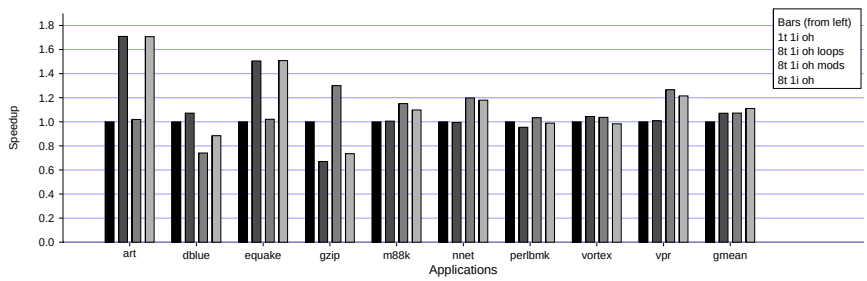


Figure 8.6: *Speedup with thread-management and communication overheads. 8-way machine, and single-issue processors.*

Figure 8.7 shows execution time breakdown for the simulations from Figure 8.6. The percentages on the vertical axis represent the number of cycles used in total to execute the simulation samples; the version of Art with loop-level parallelism uses in total about 70% more cycles as the sequential version of Art. The cycles in the parallel version are distributed over several processor cores, so the absolute execution time is still shorter than the sequential version, as indicated by the speedup reported for Art in Figure 8.6. The height of the bar thus reveals how many extra cycles are used to complete the execution in TLS mode, the *total overhead*. The *Used/Execution* and *Used/Stall* bar sections show the number of cycles during which committed threads issued instructions, or were stalling, respectively. *Overhead* represents thread management overhead cycles, and *Squashed* is execution overhead imposed by threads squashed due to misspeculations.

In this figure, one can clearly see the problem with the applications with many

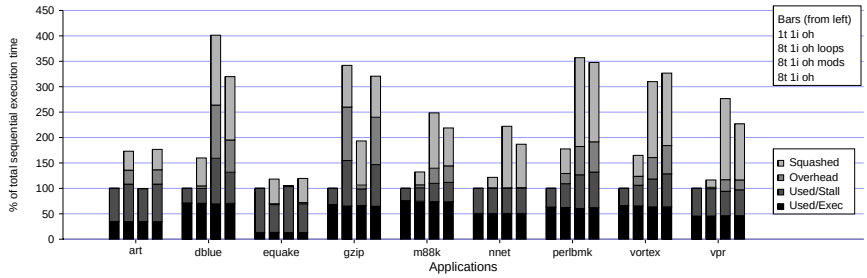


Figure 8.7: *Execution time breakdown. 8-way machine with thread-management and communication overheads. Single-issue processors.*

short threads. Both the overhead and squashed sections are large. This problem was already discovered in Chapter 3. However, in this figure one can also see that the stall times increase with TLS for many of the applications. "Stall" in this figure includes all types of pipeline stalls; flushes due to thread commit, branch mispredictions, and stalls due to memory accesses.

Branch Prediction

While not a part of the communication overhead, the results in Figure 8.6 present an opportunity to comment on branch misprediction penalties.

It is reasonable to assume that the speculative version should suffer from worse branch prediction rates than sequential execution. Since the threads are spread out among several processors with private branch predictors, the time used to train the predictors will increase. Table 8.5 contains the misprediction rates for all benchmark applications with both sequential and TLS execution. The reported TLS misprediction rates are for the simulations with both loop and module threads.

The branch misprediction rates increase only slightly for three benchmarks and decrease for the other six. The lower misprediction rates are related to execution overhead; misspeculating threads train the branch predictor which result in lower misprediction rates when threads are re-executed. However, the effect does not significantly improve the execution time for those re-executed threads.

Table 8.5: *Branch misprediction rates (percent) for sequential and TLS execution.*

App	art	dblue	equake	gzip	m88k	nnet	perl	vortex	vpr
Seq %	6.22	15.2	8.62	8.35	15.6	0.99	36.7	11.9	18.2
TLS %	6.08	16.8	11.0	7.35	14.2	1.20	27.8	12.0	13.1

In summary, the TLS effect on branch prediction is not dramatic for most benchmarks. A few applications see slightly higher misprediction rates but it does not seem to influence the performance significantly. In many other cases, misspeculating threads train the branch predictor resulting in a lower total misprediction rate, but the effect is not large enough to result in a significant reduction of wrong-path instructions in committed threads.

Memory Accesses

The memory stall cycles serviced from different levels in the memory hierarchy are shown in Figure 8.8. This figure makes it possible to better see the effects of memory latencies and communication overhead between threads running on different processors.

The vertical axis shows the total number of stall cycles for all processors in the parallel execution as a percentage of the total stall cycles in the single core used in the sequential case. Each bar is divided into five segments. The first segment, which is not visible for most simulations, is instruction stall. The warmed caches together with instruction prefetch make instruction stall a negligible fraction of memory stall. The remaining sections show data access stalls in local L1 caches, for hits in remote L1, L2, and finally main memory. Local L1 cache hits are not included in these stall times. However, the model assumes some delay when copying data from a block belonging to another thread, or updating the version of a block for a restarted thread. Therefore, some of the benchmarks may have a small amount of stall in the category local L1. Module parallelism for vortex shows visible fractions of both instruction and local L1 stall.

All applications show more total stall than the sequential case, which is expected. While the total available L1 cache space is larger on the CMP than a single core, this potential advantage is offset by communication between threads residing in different local caches, and communication overhead due to re-execution. The section remote L1 is dominated by data sharing between speculative threads. When a tread requires a copy of a cache block from a less speculative thread, it will most likely be found in another L1 cache. Another source of remote L1 hits are threads that are re-executed on another processor than in the original execution. Some old but clean blocks may be moved from the old to the new L1 cache.

Deltablue, M88ksim, Neuralnet, and Perlbmk are shown separately below the other applications due to their very high increases in memory latency. They have a very high amount of remote L1 stall. While the increase may seem remarkable, the impact is not as large as one might imagine. The reason is that the stall is measured as percentage increase against sequential execution, and in the sequential case these applications hardly have any cache misses at all. Their working sets fit in the L1

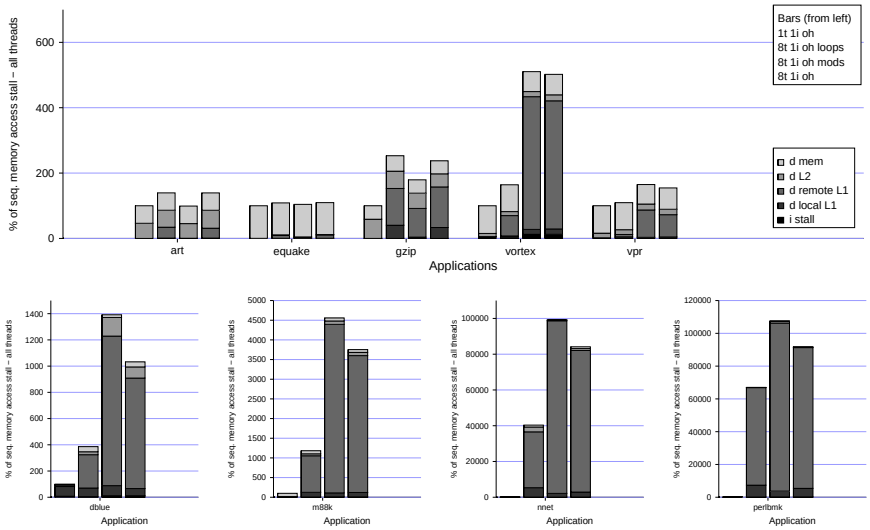


Figure 8.8: *Data stall time breakdown. 8-way machine with thread-management and communication overheads. Single-issue processors.*

cache and are already in the warmed cache. For instance, in a sample for Neuralnet the sequential version had a total of 30 dL1 misses and 2 L2 misses for data in 25 million executed instructions. Comparing these stall times with the fraction of total pipeline stall shown in Figure 8.7 gives a better understanding of what this means in real-world performance. It seems clear that there is an impact for Deltablue, M88ksim and Perlbmk, since the stall time for committed threads increases in the TLS version compared to sequential execution.

For Neuralnet *Used/Stall* does not visibly increase. The added stall is still small compared to the impact of the 3-cycle L1 load-to-use latency. Neuralnet differs from the other benchmarks as it has a very high frequency of memory accesses to a number of arrays. The frequent accesses result in problems hiding even the small L1 latency. There are simply no other instructions to schedule in between memory accesses. However, the small arrays easily fit in the L1 cache, so despite the many memory accesses there is almost no communication beyond the L1 cache. In addition, a large fraction of threads in Neuralnet are squashed, so much of the remote L1 stall time is part of the *Squashed* category.

It appears as if the total stall in the memory category is lower for TLS than sequential execution in Vortex and Vpr. This is misleading, the number of data memory accesses are the same; however, in these applications many threads are squashed

while stalling for memory, which means the completion of the memory access is overlapped with restart and re-execution. Therefore, the remaining latency does not get included as memory stall time for any specific thread and consequently does not show up in this figure.

Overall, the impact of communication overhead does not diminish the usefulness of TLS compared to the measurements with no communication overhead. On the contrary, several applications show better speedup in the simulations with a detailed memory system due to memory-level parallelism. Some applications, especially Deltablue and Gzip, are affected by the communication latencies between threads running on different cores. These applications would benefit from even faster inter-thread communication.

8.4 Sources of TLS Speedup

In this section, speedup is divided into three sources. Module-level parallelism has been investigated in earlier chapters, but here I take a closer look at loop-level and memory-level parallelism. Memory-level parallelism emerges as a source of parallelism when adding a detailed memory hierarchy, since the speculative threads will fetch data in parallel with the non-speculative thread. This either means that memory accesses from different threads overlap in a manner not possible for a single sequential thread, or that threads which are squashed effectively work as prefetch threads, reducing the memory stall for subsequent threads.

8.4.1 Module-, Loop-, and Memory-Level Parallelism

Figure 8.9 shows the contribution to speedup from different sources of parallelism. For applications with speedup, this speedup is divided into three parts depending on its source. Module parallelism if the speedup is due to parallel execution of module-level threads, loop parallelism if there is overlap in the execution of loop iterations, and prefetch if there is no overlap of executing code, but a reduction in memory stall for committed threads due to prefetching. Speedup due to increased memory-level parallelism in successful threads compared to sequential execution is, however, not shown separately but as part of module- or loop-level parallelism.

Loop-level parallelism dominates for two applications: Art and Deltablue. Module-level parallelism is the major source of parallelism for three benchmarks: Gzip, M88ksim, and Neuralnet. In Equake, almost all of the parallelism is due to prefetching from squashed threads, and for Vpr both module parallelism and prefetching contribute.

Comparing the result for Art in Figure 8.5 with the speedup in Figure 8.6 the speedup is higher with communication overhead, despite the fact that there seems to

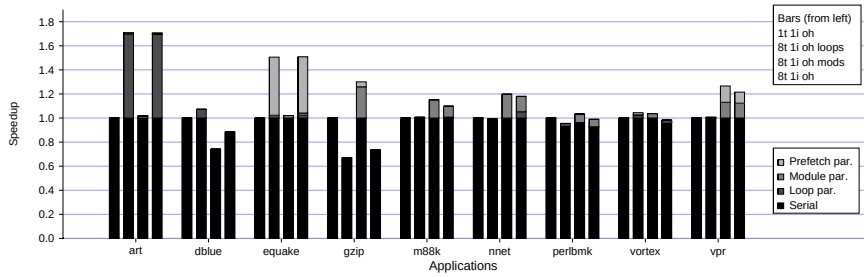


Figure 8.9: *TLS speedup resulting from loop or module parallelism, and prefetching effects. 8-way machine with thread-management and communication overheads. Single-issue machine.*

be no noticeable prefetching effect. This is due to an increase in loop-level parallel overlap, which originates from memory-level parallelism in successful threads. Art is memory-intensive, which makes this effect noticeable. In addition, the problem with short threads imposing a large fraction of thread-start overhead is reduced since the execution time for each thread is extended due to frequent memory access stalls with the detailed memory hierarchy.

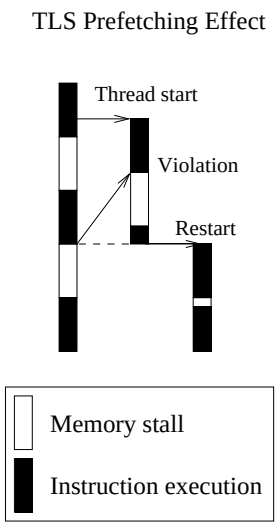


Figure 8.10: *The prefetching effect of speculative threads.*

Figure 8.10 illustrates the prefetching effect of failed speculative threads. A speculative thread is spawned and runs for some time. During that time, there is a long latency memory access (i.e. an L2 miss).

The thread is restarted due to a dependence violation. When the thread is restarted and run for the second time, the cache line is already fetched and the second execution will proceed faster, as illustrated by the shorter memory access section in the figure. In fact, other threads may also benefit from the prefetch.

Figure 8.11 gives another perspective on the memory hierarchy effects. This figure is the same as Figure 8.8 in all respects but one – only memory stall experienced by threads that commit is included, memory stall for squashed threads is filtered out. Notice that for Equake and Vpr, the total stall time is reduced compared to sequential execution. For Equake and Vpr, almost half of the memory stall cycles are removed. Unfortunately, for Vpr the module-level speculation used to achieve this high memory-level paral-

lelism also adds a fair amount of remote L1 stall time, so in the end the gain is closer to a one fourth reduction of total stall cycles for this application.

All three source of parallelism turn out to be important, but in general only one or perhaps two sources are applicable for an individual application. The effect of memory-level parallelism should not be neglected since it can be an important contributor to the overall speedup gained from TLS.

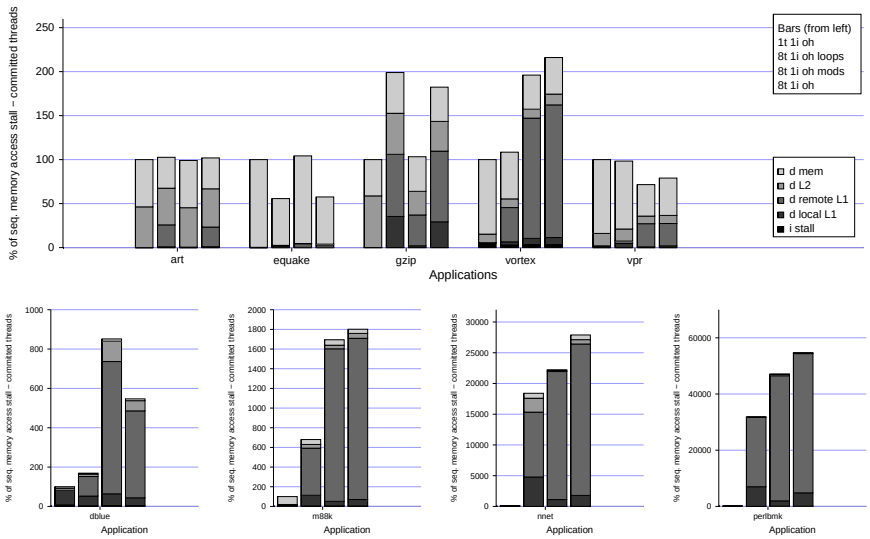


Figure 8.11: *Data stall time breakdown for committed threads. 8-way machine with thread-management and communication overheads. Single-issue processors.*

8.4.2 Multiple-Issue Processors

There is another source of parallelism exploited in most modern processors: instruction-level parallelism. So far, all processor models in this thesis have been single-issue. With wide-issue out-of-order processors, threads can execute faster and hide memory latencies better. As mentioned earlier, one may expect this to result in less gain from TLS.

In Figure 8.12 results with single-issue processors are compared to 2, 4, and 8-issue processors, configured according to *model 5*. For each application, the best performing alternative of module-level, loop-level, or both types of parallelism has been selected for this graph. The leftmost bar for each benchmark shows the result for single-issue processors and as we move to the right the issue-width is increased.

All speedups are computed as the relative speedup of an 8-way CMP compared to a single-core processor, where the single-core processor and CMP has the same issue width. Therefore, differences in speedup are not due to ILP, but rather due to differences in exploitable thread-level parallelism, to show how well TLS can speed up execution in an environment with wide-issue processor cores.

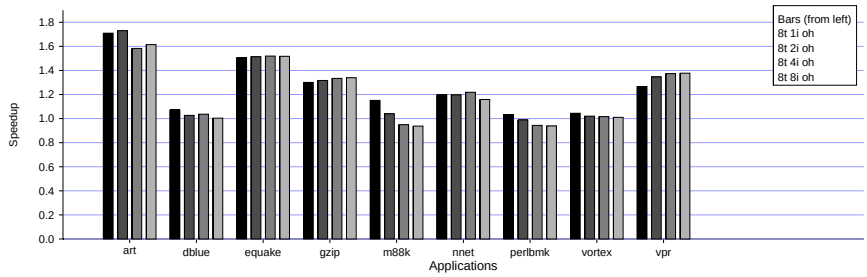


Figure 8.12: *Speedup with multiple-issue processors. 8-way machine with thread-management and communication overheads. 1, 2, 4, and 8-issue processors.*

Somewhat surprisingly, the results do not change very much for most applications. M88ksim is the most negatively affected application. It has high ILP and also relatively short speculative threads, exacerbating the loss of potentially exploitable TLP. The change of CPI in M88ksim from single-issue to 4-issue processors is shown in Table 8.6. The CPI numbers in Table 8.6 are for simulations with both module- and loop parallelism and includes instructions executed by squashed threads. For applications with prefetching the CPI for committed threads only is lower than for all threads due to less memory stall.

Table 8.6: *Comparison: CPI for single-issue vs 4-issue processors.*

App	art	dblue	equake	gzip	m88k	nnet	perl	vortex	vpr
1i/seq	2.93	1.47	8.59	1.49	1.34	2.02	1.75	1.54	2.64
1i/TLS	3.13	1.75	7.18	1.73	1.51	2.03	2.21	1.89	2.35
4i/seq	2.70	1.04	8.52	1.14	0.77	1.74	1.41	1.08	2.46
4i/TLS	2.97	1.47	7.03	1.50	1.16	1.77	2.05	1.58	2.14

It is clear that the gain in ILP for sequential execution going from single-issue (first row in the table) to 4-issue (third row) is significant. The ILP gain with TLS enabled (second and fourth rows) is smaller, which indicates a trade-off between exploiting ILP and TLP. Perlbmk and Art also gain some performance from ILP at the expense of less exploitable TLP, while for the other applications taking advantage of ILP does not prevent thread-level parallelism from being exploited as well. Equake

is memory-bound and gains very little from ILP. Vpr and Gzip even enjoys some gain in thread-level parallelism with wider-issue processors. The total number of threads started is lower with wide-issue processors for both applications, leading to lower total overhead, but the parallelism is not negatively affected. The underlying reason for this phenomenon is unclear.

For completeness, the full set of simulations for module and loop parallelism with 2, 4, and 8-issue processors are presented in Figure 8.13. The corresponding results for single-issue processors were shown in Figure 8.6. In the following sections, the baseline CMP will use 4-issue processors.

To summarize, the impact of multiple-issue processors is, for most applications, not negative in terms of lower thread-level parallelism. For memory-bound applications the CPI is not affected much by multiple-issue, and for many applications exploiting ILP and TLP seems largely orthogonal. For some applications though, typically programs with a fair amount of ILP, there is indeed a trade-off between ILP and TLP.

8.4.3 Deferred Squash

In the previous section, it was apparent that for some applications, prefetching is the dominant or even only source of thread-level parallelism. No useful overlap of execution exists. The speculative threads are, however, still restarted every time there is a dependence violation. When the thread is restarted, it will not continue to prefetch data. If there is in fact no execution overlap, repeatedly trying to restart the thread is just wasteful. The same code might be executed repeatedly in vain. One proposed solutions to this is to prevent the thread from running again, which was discussed in Chapter 6.

The prefetching effect gained by squashed threads is similar to runahead execution (investigated by [DM97, MSWP03] among others). In both cases, a thread running ahead of normal execution will execute memory accesses in advance of the main thread, thereby reducing the effective memory latency. In addition, the results produced by these advance threads may or may not be correct, and thereby must be stored separately, not able to alter architectural state. A main difference between runahead threads and TLS is that the results of a speculative thread can be used and merged with architectural state if found to be correct. The results produced by a runahead thread are always thrown away, and the code is re-executed by the main thread. The upside is simpler hardware support.

Since the results for runahead threads can not be used, the objective is simply to keep them running far enough ahead of the main thread to provide useful prefetching. For TLS, an observation is that if a thread contributes more to performance by prefetching data than with actual parallelism, it could be more useful to allow it to

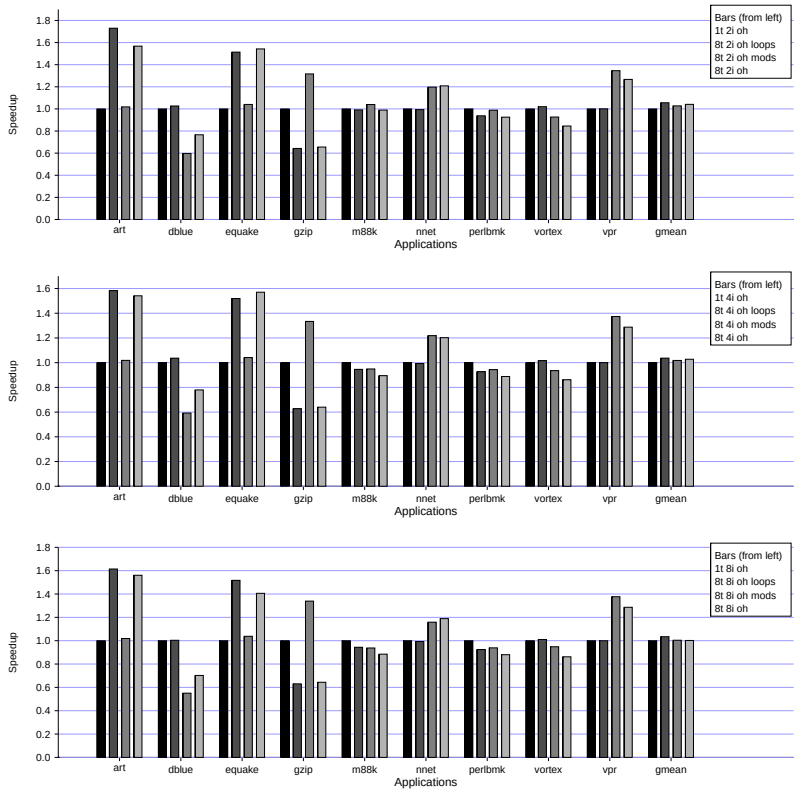


Figure 8.13: *Speedup with multiple-issue processors. 8-way machine with thread-management and communication overheads. 2, 4, and 8-issue processors.*

continue executing, i.e. prefetching, even after a dependence violation.

Chen et al. [CSL03] investigate such a policy for the superthreaded architecture, *wrong thread execution*. When speculative threads are aborted, they are marked as wrong-path threads but allowed to continue until the thread reaches its final instruction. Then, they are squashed instead of committed. They find an average performance improvement of 9.3%, with the best application improving 18.5% due to the reduction in cache misses, and with a reasonably small overhead. They use a special *wrong execution cache* to store such prefetched data in order to avoid cache pollution. The superthreaded architecture is not data speculative but control speculative. This means the wrong-path threads they allow to continue may not be needed later, so there is indeed a risk of cache pollution.

The architecture investigated in this thesis is data speculative, and therefore the

situation is somewhat different. Module threads are not control speculative. Once the function call is encountered, the continuation is certain to be executed, and this is typically the case for loop iteration threads as well.[2] Therefore, the risk of cache pollution should be small. Consequently, in my implementation there is no added cache or indeed any extra hardware. The only alteration is that after a dependence violation, misspeculating threads are marked for *deferred squash* and allowed to continue executing. The thread is allowed to run either until it reaches its last instruction, or until it becomes the head thread, whichever occurs first. A deferred thread may not spawn new threads of its own.

As opposed to the technique in [CSL03] the thread is not aborted when it reaches its last instruction, instead it is restarted. Their scheme also terminate wrong-path threads if the processor is needed for a new speculative thread. My deferred threads are not squashed to make room for new speculative threads. However, I leave an investigation of which of these policies work better for future work.

Figure 8.14(a) shows results with deferred squash. It is clear that deferred squash will improve the speedup in some cases. More specifically, the speedup is improved for Equake and Vpr, the threads which showed virtually no parallel overlap, but significant gains due to the prefetching effect. Vortex is another application which gains from deferred squash.

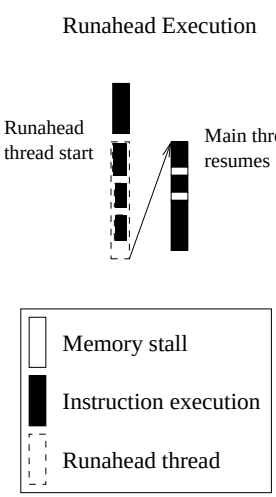
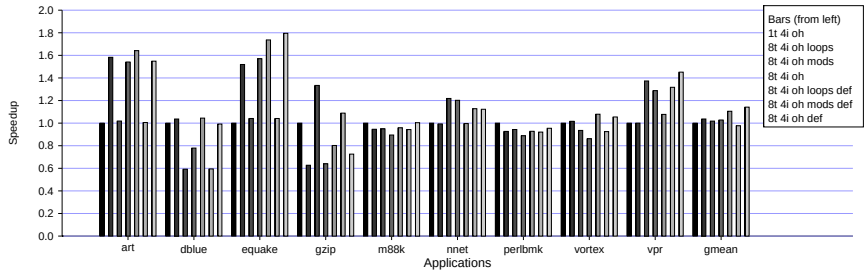


Figure 8.15: *A runahead thread.*

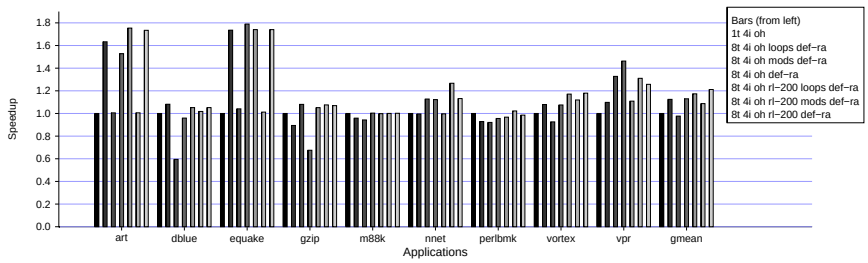
There are several variations of runahead threads and precomputation. Runahead threads as described by Mutlu et al. [MSWP03] begin where the main thread gets stalled, as opposed to a TLS thread which begins farther ahead in the sequential instruction stream. In addition, the runahead threads does not wait on long-latency memory accesses; that way a single runahead thread may prefetch many locations from memory concurrently, increasing the memory-level parallelism.

Runahead threads are illustrated in Figure 8.15. When the main thread is interrupted by an L2 cache miss, the execution continues as a runahead thread. While the miss is being serviced, the runahead continues to execute and issue several more memory requests, but does not wait for their completion. When the original request is satisfied the main thread is restarted, but the memory

[2] In a real implementation, loop threads may be control speculative. As explained before, however, my simulator does not capture this. There are some rare exceptions to the rule for module threads too, also mentioned earlier.



(a) *Deferred squash.*



(b) *Deferred squash with runahead mode.*

Figure 8.14: *Speedup with deferred squash on an 8-way CMP with 4-issue processors.*

access latencies for future accesses are reduced.

A speculative thread, even when marked with deferred squash, works just like an ordinary thread. That is, it stalls on cache misses, waiting for the correct data to arrive before continuing. Figure 8.14(b) shows the results for a variant of deferred squash which mimics the behavior of a runahead thread. In deferred mode level two cache misses issue a memory request, but the processor does not wait for the requests to finish. Instead, it will continue executing the thread without the requested data. I call this *deferred squash with runahead mode*.

Art, which did not benefit from deferred squash alone, does benefit sligthly from the runahead variant, expecially when combined with run-length prediction. However, in general there is no additional gain. The graph shows results both with and without combining this technique with run-length prediction. For Vortex, the combination of deferred squash with runahead mode and run-length prediction does somewhat better than any of the techniques alone. Run-length prediction is more closely investigated in the next section.

It should be noted that there is a source of error in these simulations. In my trace-based methodology, the code is not actually executed during simulation; therefore,

execution errors due to the misspeculations are not visible. In a real machine, this could lead to erroneous address calculations which means either the wrong addresses are prefetched or there is a memory access exception. Another possibility is that execution could take the wrong path due to the misspeculation, therefore not prefetching the desired data. In runahead mode, all level two cache misses are requested but then ignored. This is another effect not taken into account in the simulation; it will appear to the simulator as all fetches have been completed, even if they are actually still outstanding requests. The same sources of error as for deferred squash are the possible repercussions, but even more likely since there may be multiple locations with erroneous contents.

One should also remember that my simulator does not model congestion in main memory accesses. In a real memory system, some accesses may experience additional stall due to congestion for bandwidth demanding applications. This is another reason why the measured benefit of prefetching and especially runahead mode may be optimistic.

Due to these sources of error, the gain reported due to prefetching with deferred squash and deferred squash with runahead mode is probably optimistic. Therefore, this experiment should be interpreted as a study of the potential of these techniques, rather than an exact evaluation of the performance gain one can expect.

Two things are clear after this investigation. First, deferred squash is useful for applications where the prefetching effect is a factor contributing to speedup. Second, for some applications traditional TLS works better. Therefore, deferred squash should be implemented with care. It may be more useful together with some technique measuring the prefetch gain and selectively using traditional TLS or deferred squash whenever either variant works best. In addition, the policy used by Chen et al. [CSL03] – aborting wrong-path threads when the processor can be used for new speculative threads – could potentially be a better alternative for applications where there is also module- or loop-level parallelism. However, an investigation of these alternatives is beyond the scope of this thesis and left for future work.

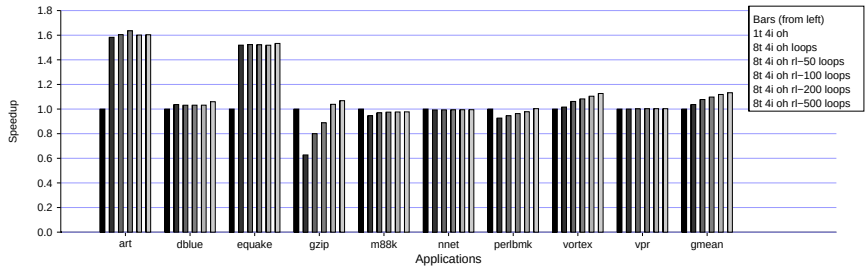
8.5 Run-Length Prediction Revisited

Run-length prediction (*model 6*) was introduced in Chapter 4. It was shown to be a useful technique to dynamically filter out threads that are so short they are unlikely to contribute any useful parallelism. The simulation methodology used, however, did not consider communication overhead or out-of-order execution. In addition, only module-level parallelism was used.

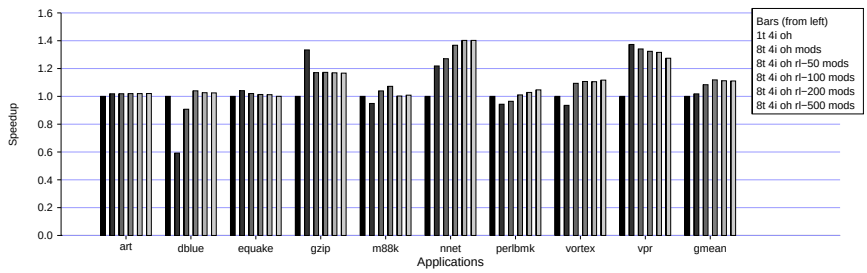
With the detailed simulation model, run-length prediction has been extended to loops. For loops, the predictor provides the added benefit of automatic unrolling; when a loop iteration is shorter than the run-length threshold, a new thread is only

started every *n*th thread, where *n * iteration run-length* exceeds the run-length threshold.

Figure 8.16(a) and Figure 8.16(b) show the performance for loop- and module-level parallelism respectively. For each benchmark, run-length thresholds of 50, 100, 200, and 500 cycles have been used. Looking at the mean speedups, it appears that overall a threshold of 500 cycles gives the best performance for loops, and 100 or 200 cycles for modules with these machine parameters.



(a) *Impact on loop-level parallelism.*



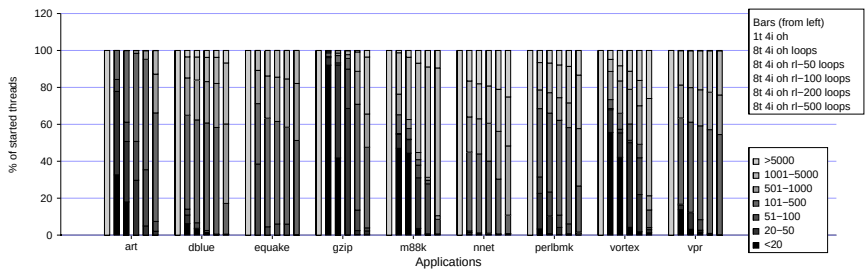
(b) *Impact on module-level parallelism.*

Figure 8.16: *Speedup with run-length prediction. 8-way machine with thread-management and communication overheads. 4-issue processors.*

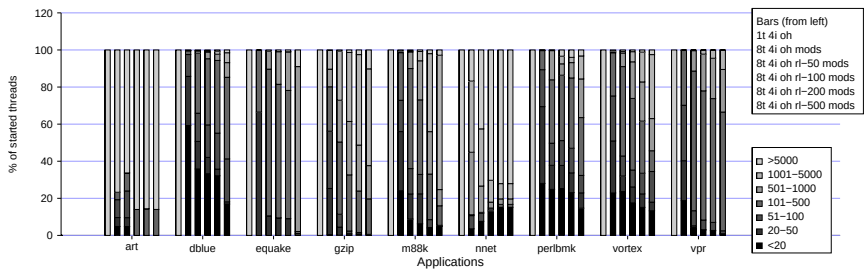
Art, with plenty of loop parallelism sees a slight speedup with run-length prediction. The other loop-heavy application, Equake, shows no difference, which is due to the fact that most loops are already above the threshold, and besides, most loops misspeculate; the performance improvement for Equake comes from prefetching. M88ksim and Perlbmk see some improvement for loop parallelism but still do not achieve any speedup. Vortex, however, enjoys a healthy speedup improvement. Most remarkable of all, Gzip, which showed a major slowdown, actually manages a slight speedup for loop-level parallelism with run-length prediction.

For modules, the performance of Deltablue has a similar remarkable recovery. Vortex, Neuralnet and M88ksim also benefit significantly. Vpr loses some performance. This is not totally unexpected, since a large part of its speedup comes from prefetching. For Gzip, unfortunately, run-length prediction hurts the performance.

The aim of run-length prediction is to filter out short threads, thereby reducing the amount of overhead and threads so short they are unlikely to contribute any significant parallelism. The thread size distributions in Figure 8.17 show that this is exactly what happens both for loops and modules. The second bar from the left in each cluster show the distribution without run-length prediction. Then, as the thresholds are increased in the third to fifth bar for each benchmark, the number of short threads gradually decrease.



(a) *Impact on loop-level parallelism.*



(b) *Impact on module-level parallelism.*

Figure 8.17: *Thread-size breakdown. Run-length prediction on an 8-way machine with thread-management and communication overheads. 4-issue processors.*

Note that the graph shows the fraction of threads in each size category. For Neuralnet, it might seem the number of very small threads increases as the threshold is increased. This is not the case. Instead, the number of medium-sized threads decreases, making the below-20 category a larger fraction of a smaller total number of

threads. Unfortunately, the predictor fails to filter out many of the smallest threads for this application. Simulations with single-issue processors show a similar result. In fact, the improvement in speedup is even slightly better with these less complex processors.

To sum up, run-length prediction works well even with communication overhead and multiple-issue processors. Many of the short threads are filtered out, resulting in fewer threads, less overhead, and better speedup. With the overhead and memory system parameters used in these simulations, a run-length threshold of 200 works well in almost every situation. There remains a few percent slowdown for two applications with loop-parallelism, but for all other cases slowdowns were eliminated.

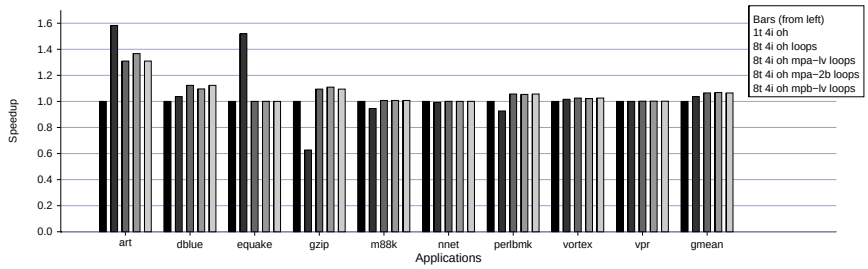
8.6 Misspeculation Prediction Revisited

Similar to run-length prediction, misspeculation prediction (*model 7*) presented in Chapter 6 is re-evaluated with the detailed simulation model. Figure 8.18 shows misspeculation prediction for loop and module parallelism respectively. Three versions of misspeculation prediction are evaluated, last-value and 2-bit type A predictors, and a last-value type B predictor.

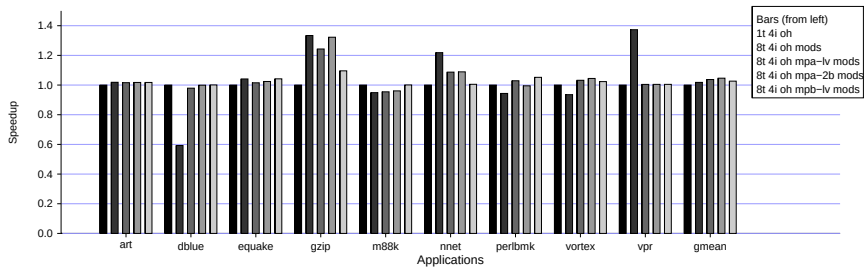
The results are mixed. For Equake with loop parallelism, and Vpr with module parallelism, all speedup is wiped out. This is because these applications gain much from prefetching. Vpr does have some module parallelism as well, but if this originates from functions which misspeculate occasionally, the misspeculation predictor will disable speculation for these threads as well. The fundamental problem is that memory-level parallelism is not a criterion for when to disable speculation. Only misspeculations are taken into account.

For module parallelism in Neuralnet and loop parallelism in Art, the performance is also significantly reduced, while other applications, e.g. the loop versions of Deltablue, Gzip, Perlbmk, and Vortex, gain from misspeculation prediction even with the detailed model. The applications which gain performance are primarily those with many (typically small) threads, and much thread-management and execution overhead. For module speculation, only Perlbmk and Vortex benefit, and the slowdown is at least removed for Deltablue. Clearly, run-length prediction does a much better job judging by performance.

A primary motivation for misspeculation prediction, however, was to reduce excessive overhead. Figure 8.19 shows that the technique still works in this regard. For applications which lose some parallelism, e.g. loop parallelism for Art and module parallelism for Gzip and Neuralnet, the upside is almost no remaining overhead. In some cases, e.g. loop parallelism for Deltablue, Gzip and Perlbmk, the overhead is removed while increasing the speedup.



(a) *Impact on loop-level parallelism.*

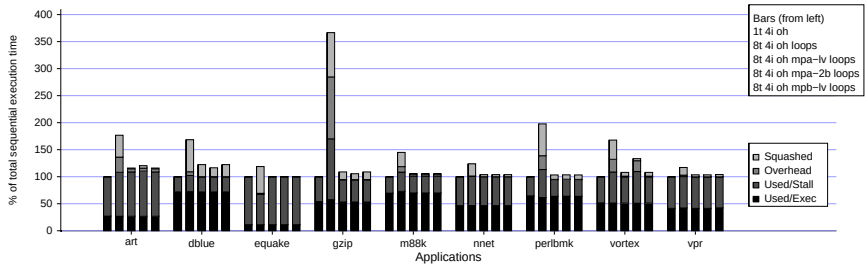


(b) *Impact on module-level parallelism.*

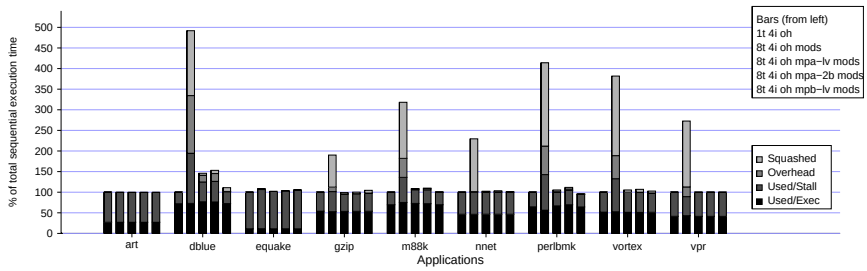
Figure 8.18: *Speedup with misspeculation prediction: 8-way machine with thread-management and communication overheads, 4-issue processors.*

A contributing factor to why the type A misspeculation predictor does not work as well for modules is the problem discussed in Section 7.3.4, i.e. that when TIDs are used to keep track of threads it is not always possible to find the common ancestor. This problem is significant in some of the applications, for instance Deltablue, Perlbmk and Vortex where the correct ancestor is not found most of the time. However, even if the common ancestor is often not found it seems like most of the misspeculating threads are filtered out. Only Deltablue has a sizable portion of remaining overhead, but even for this application most overhead is gone.

Misspeculation prediction still does its job of removing excessive overhead with the detailed simulation model, and improves the performance for some applications. Unfortunately, a big problem is that memory-level parallelism is not taken into account. Intuitively, it seems possible to reduce this problem by adding prefetching as a second criterion when deciding which threads to classify as non-speculative. For instance, one idea would be to use a counter keeping track of the long-latency misses, or alternatively the total memory stall time for each thread. If, after re-executing the thread following a misspeculation, the long-latency misses or stall time is sig-



(a) *Impact on loop-level parallelism.*



(b) *Impact on module-level parallelism.*

Figure 8.19: *Execution time breakdown. Misspeculation prediction on an 8-way machine with thread-management and communication overheads. 4-issue processors.*

nificantly reduced, the misspeculation prediction would be ignored. Evaluating such extensions is beyond the scope of this thesis.

8.7 Related Work

Runahead execution and precomputation are similar to TLS in the sense that both techniques execute code ahead of a main thread in the instruction stream. The difference is that precomputation only uses this for prefetching and training branch predictors; no computational results from the advance thread are used. However, the main thread, which performs all useful computations, is sped up due to fewer cache misses and branch mispredictions. This is related to the prefetching effect of speculative threads observed in this chapter.

The advantage of runahead execution compared to TLS is that it requires less hardware support and has lower complexity, as there is no need for dependence detection or roll-back. In addition, the runahead thread can use the same proces-

sor as the main thread. Runahead execution has been investigated by Dundas and Mudge [DM97] and Mutlu et al. [MSWP03]. Many other techniques use helper threads executing ahead of the main thread in order to reduce miss penalties, for instance Assisted Execution [SD98], SSMT [CSK+99], Speculative Precomputation [CWT+01], and software-controlled pre-execution [Luk01].

The importance of memory-level parallelism for TLS is stressed by Liu et al. [LTS+06]. Their POSH TLS compiler uses profiling to try to discover threads gaining much from prefetching. This metric is one criteria the compiler uses when deciding where to create spawn points for speculative threads. As mentioned, Chen et al. [CSL03] also investigates the prefetching effect of misspeculating threads.

The impact of memory latencies and issue-width has recently been investigated by Ohsawa et al. [OTKM05] for their Pinot architecture. However, except for commenting on how this affects speedup, they do not analyze the results more closely.

8.8 Conclusions

In this chapter, I have investigated the performance impact of a number of machine dependent parameters. In addition, run-length prediction and misspeculation prediction has been re-evaluated with a detailed machine model. Finally, TLS with deferred squash and runahead mode has been introduced.

The main findings are:

- Issue-width does not affect the speedup for most applications; ILP is often orthogonal to speculative TLP, or not significant enough to affect the TLS negatively. However, for some applications with plenty of instruction-level parallelism and short threads, there is a trade-off between exploiting ILP and speculative TLP.
- Memory access latency typically does not have a negative impact on speedup compared to the the experiments with inherent parallelism. On the contrary, memory-level parallelism and the prefetching effect from squashed threads contribute to higher speedups when taking communication into account.
- Evaluation of the potential of deferred squash and deferred squash with runahead mode. It is shown that some applications benefit more from the prefetchine effect than TLP; for those applications, deferring the squash after a misspeculation can increase the benefits of the prefetchine effect.
- Run-length prediction does work as intended with communication overhead taken into account, and also with multiple-issue processors. In addition, run-length prediction for loops is extended to work as a dynamic loop unrolling mechanism. This technique is also shown to be efficient for some applications.

9

Simultaneous Multithreading and TLS

As discussed in Chapter 2, TLS is possible to implement on any shared-memory architecture supporting multiple threads. This chapter extends the exploration of the TLS design space with simultaneous multithreading. On an SMT processor with TLS support, new threads will start on the same core as the parent thread. On a hybrid TLS chip multiprocessor with SMT cores, a new thread could start either on the same core or another core. Since the threads share the same core, and typically L1 cache, threads starts and restarts can potentially be more efficiently implemented on an SMT processor. This translates into lower thread management overhead. On the other hand, the threads share execution units and cache, which might impact the performance negatively. The results show that, given an equal total issue width for the SMT processor and the cores in a CMP, the SMT processor typically performs better due to the lower thread management overheads, as well as lower inter-thread communication costs.

A second machine model investigated in this chapter is a TLS machine supporting only one speculative thread at a time. This also means a processor, or SMT thread, can not begin to execute a new thread before the current one has committed; preemption of idle tasks is not supported. I examine the performance potential for this machine, and discuss possible simplifications to the TLS implementation. Surprisingly, this machine model can successfully exploit a substantial part of the parallelism available with the dynamic TLS techniques used in Chapter 8.

9.1 Simulation Methodology

Support for SMT processors is available in the simulation model described in Chapter 7. In this chapter, SMT processors with two and four threads per core are compared to CMP models with a similar amount of resources. The parameters for all machine models used in this chapter are summarized in Table 9.1.

Table 9.1: *Baseline machine parameters - SMT and CMP models.*

Feature	*Parameters*		
	2-issue	*4-issue*	*8-issue*
Fetch/issue/commit width	2/2/2	4/4/4	8/8/8
ALUs per core	2	4	8
Load/Store units per core	1	2	3
Floating point units per core	1	2	3
Instruction window size	128	256	384
LSQ size	48	64	96
Branch predictions per cycle	2	2	2
Pipeline length – integer	8	8	8
Pipeline length – floating point	11	11	11
Branch predictor	G-share, 16k predictor table, 8-bit global history, 2k target buffer, 8 entry Return address stack per thread.		
L1 caches	32+32 kbytes i+d, 4-way associative 3-cycle load-to-use latency 3 extra cycles for version upgrade or block duplication from less speculative thread.		
L1 instruction cache	Sequential prefetching		
Shared L2 cache	4 Mbytes 32-way associative 15-cycle total latency without contention		
Cache block size	32 bytes		
On-chip bus	256 data bits, 2 cycle latency		
Main memory	200 cycle latency		
SMT thread-start overhead	10 cycles (no bus activity)		
SMT restart overhead	5 cycles (no bus activity)		
SMT commit overhead	5 cycles (no bus activity)		
CMP thread-start overhead	20 cycles + 4 bus transactions		
CMP restart overhead	10 cycle + 1 bus transaction		
CMP commit overhead	10 cycles + 4 bus transactions		

The machine parameters are the same as in Chapter 8 except for thread-management overhead, which should be lower for an SMT considering that threads share the same core; there is no need to transfer initial or altered register values through the memory system and on-chip bus. Instead, the SMT core can be slightly modified to support fast copying of registers to the new thread. Such mechanisms have been proposed for the DMT [AD98] and IMT [PV03] architectures. The thread management overhead parameters for SMT are reduced to reflect this.

The simulator is not modified for the experiments with a single speculative thread. However, there is a possibility to limit the number of available thread contexts. In the previous chapter, this number was set high enough so that the simulations never ran out of contexts. For the simulations with one speculative thread, the number of thread contexts is set to two. This means there can only be one non-speculative and one speculative thread in the system at the same time. When the speculative thread has finished executing, it must wait to become head-thread before it can commit, and during this time there are no free context which can be used to start new speculative threads. Care has been taken to make sure TID allocation does not affect the results, as TIDs are not necessary in a machine with only one speculative thread.

Interpreting the Figures

Table 9.2 provides a reference for the abbreviations used in the figure legends in this chapter. Note that CMP models, like in the previous chapter, are not explicitly labeled in the legends; whenever the legend does not state than an *smt* model is used, the results are for a CMP. Also, remember that the legend for number of thread contexts is not used until Section 9.3.

Table 9.2: *SMT and single speculative thread figure legends.*

Legend	*Explanation*
*y*t	Maximum number of running threads (t) (y = 1, 2, 4, or 8).
loops	The keywords *loops* or *mods* show that only loop-level
mods	or module-level threads are used. Default is both kinds enabled.
oh	Thread-management and communication overhead (oh).
*x*i	Issue width (i) of the processors (x = 1, 2, 4, or 8).
rl-*s*	Run-length predictor with threshold *s*.
smt-*z*	Number of threads per SMT core (z = 2, 4).
	If no smt number is given there is only one thread per core.
*m*c	Maximum number of available thread contexts (m >= cores/SMT-threads).
	If m is not given, it is assumed large enough to be a non-issue.
gmean	Geometric mean.

9.2 TLS With Simultaneous Multithreading

The main differences, from a performance standpoint, of starting and running two threads on different hardware contexts within an SMT core or different cores on a chip multiprocessor are:

- Data sharing between speculative threads is handled through the local L1 cache for threads on the same core, while the data has to be transferred through the on-chip interconnect (in my model a shared bus) on a chip multiprocessor.

- Spawning threads include copying initial register values to the new thread. On an SMT, this can be handled within the processor core without major modifications to a base SMT architecture. For the CMP, this data has to go through the memory system, i.e. be written to L1 and transferred across the on-chip bus.

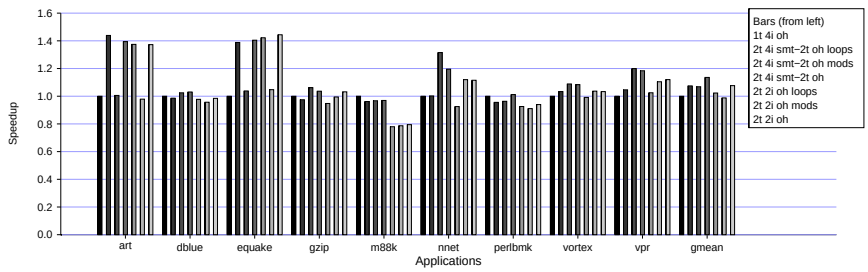
- Available L1 cache space is potentially larger for the CMP since each thread gets a private cache. In principle, the cache of the SMT could be made larger to compensate for the additional threads. However, these are several implementation issues complicating such a design choice. First, there is a trade-off between size and latency which is critical for level one caches. Second, the SMT cache needs more read ports for best performance, which further increases its size and complexity. Third, speculative versions are stored in cache ways, which means increasing the total size may not be enough. Increasing the associativity is likely to lead to increased latency.

- The threads in the SMT share execution units while in the CMP each thread has full access to all available resources. While the SMT may use a wide-issue core in order to give all threads room to execute efficiently, it is less scalable than a CMP. The complexity of wide-issue processors have so far prevented efficient implementations of very wide cores. Constructing scalable chip multiprocessors seems to be an easier task.

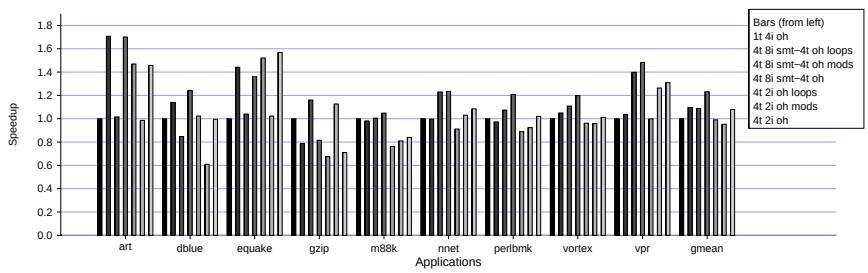
When comparing SMT and CMP architectures, I will use SMT processors with an equal total issue width as all the cores in a CMP. No detailed analysis of die space requirements for these design points have been made. However, this comparison provides a rough estimate of which architecture is the best TLS substrate with a similar amount of execution resources. While hybrid CMP-of-SMT architectures are interesting, the comparisons are between SMT-only and CMP-only models. With this comparison, the differences between the architectures can be assessed more easily.

9.2.1 Experimental Results

Figure 9.1(a) presents the results of a comparison of two designs which both support two threads. One is a 4-issue SMT, and the other a CMP with two 2-issue cores. That is, in total they can issue the same amount of instructions in each thread. Results with only loop- or module-level threads as well as a combination of both types are shown. Figure 9.1(b) shows results for a 4-thread 8-issue SMT compared to a 4-way CMP with 2-issue cores in a similar manner. The processor cores are configured according to Table 9.1.



(a) *2-thread 4-issue SMT compared to a 2-way 2-issue CMP.*



(b) *4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*

Figure 9.1: *TLS on an SMT: Comparing SMT and CMP designs with equal total issue width.*

In the previous chapter, the speedup of the TLS models were compared to sequential execution on the same type of processor core used in the CMP, e.g. TLS speedup for an 8-way 4-issue CMP was computed relative to sequential execution on a single 4-issue core. For this figure, I want to compare the performance of the CMP and SMT machines; therefore, the speedups for both machines need to be computed relative to the same sequential base machine. I have selected the issue-width of the SMT core, i.e. the results in Figure 9.1(a) are relative sequential execution on a 4-issue processor.

Note that the results for the CMP can not be compared to the speedup using 2-issue cores in Section 8.4.2 since both the number of cores and the base machine used to compute speedup differs. However, remember that one conclusion from those experiments were that a 2-issue machine can exploit much of the available ILP. Therefore, using 2-issue cores for the CMP in this comparison should not result in a totally skewed comparison due to differences in ILP.

Overall, the SMT machines seem to perform somewhat better. This is true both for memory-bound applications such as Art, and high-ILP applications such as M88ksim. This is not unexpected, since the overhead is lower in the SMT and the machines have been given the same total issue width. The SMT is not throttled by lower total fetch- or issue width.

In addition to the baseline parameters in Table 9.1, simulations were conducted with both the total size and the associativity of the level one caches on the SMT scaled with the number of threads. While the baseline machine uses 32 kbyte 4-way level one caches, the 2-thread SMT model was equipped with 64 kbyte 8-way caches and the 4-thread SMT used 128 kbyte 16-way caches. While building a 16-way level one cache might not be a realistic design point, this experiment provides a comparison of the two architectures that is not influenced by differences in cache space.

With the 2-thread SMT, the differences are hardly noticeable. For the 4-thread machines, the results are shown in Figure 9.2. The speedup for Neuralnet and Perlbmk sees a healthy increase, and several other applications execute somewhat faster.

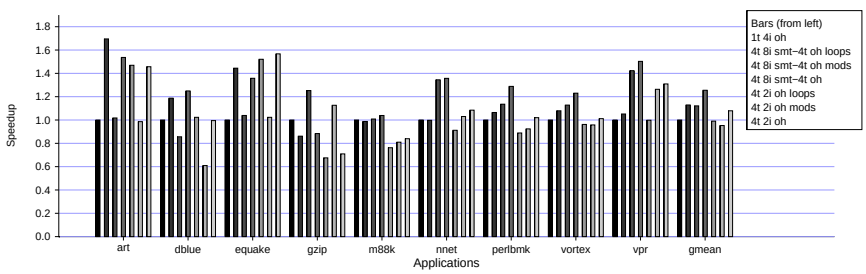
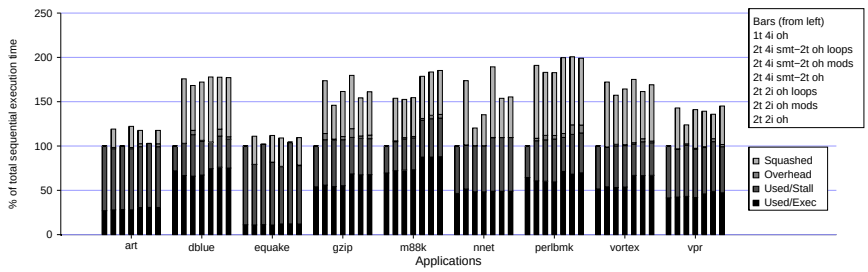


Figure 9.2: *TLS on an SMT: Speedup on a 4-thread 8-issue SMT compared to a 4-way 2-issue CMP, where the SMT has the same total L1 cache size as the CMP.*

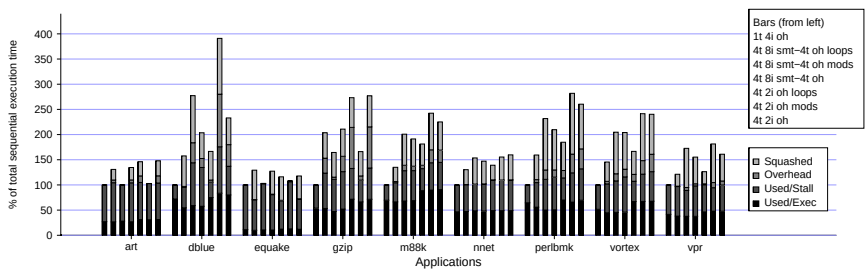
Equake is the only application where the CMP performs slightly better. Equake already has quite long threads and little overhead. Therefore, it does not gain much from the lower thread management overhead. In addition, even though Equake is memory-intensive, it does not benefit from larger L1 cache space. Figure 9.3 shows the total execution time, i.e. the sum of the execution times for all executed threads.

The *Used/Exec* and *Used/Stall* segments of the bars show the fraction of cycles where committed threads issued instructions or stalled, respectively. A close inspection of the execution time breakdown reveals that the SMT version of Equake has somewhat more stall time. This is not due to memory stall, in fact the total memory stall is slightly higher for the CMP version. However, the resource sharing in the SMT introduces some stall time in the pipeline. There will be more about pipeline stalls in Section 9.2.3.

Figure 9.3 also shows that, as expected, the thread management overhead is in general lower for the SMT models – this is especially visible in the 4-thread simulations. The execution overhead from squashed threads remains approximately the same. Experiments show that even if the issue-width is doubled for the processor cores in the CMP, the results do not change much. Most of the performance advantage for the SMT processors is due to the lower overhead, something the CMP cannot match even with wider-issue processors.



(a) *2-thread 4-issue SMT compared to a 2-way 2-issue CMP.*



(b) *4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*

Figure 9.3: *TLS on an SMT: Execution time breakdown.*

Figure 9.4 shows memory stall times for the 4-thread experiments. For many applications, the remote L1 stall is significant with the chip multiprocessor. Naturally, this overhead does not exist for the SMT. However, for a few applications like Vortex

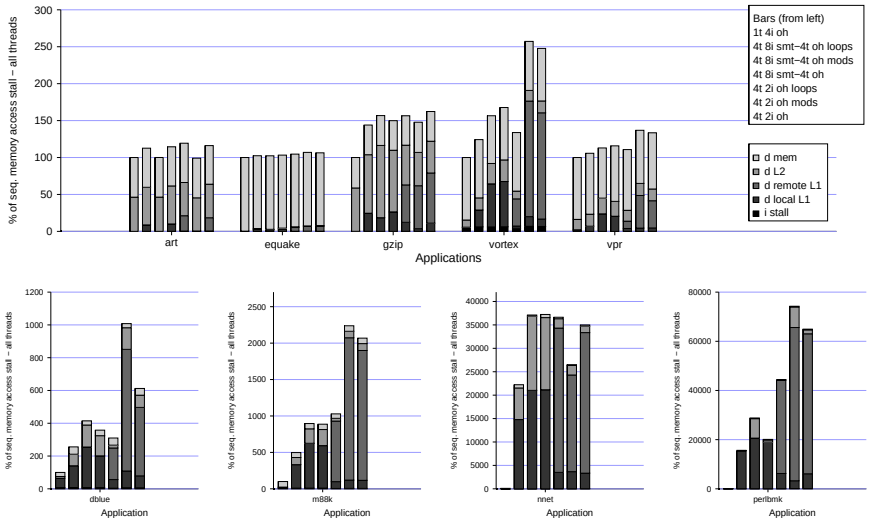


Figure 9.4: *TLS on an SMT: Data stall time breakdown for a 4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*

and Neuralnet the local L1 stall is significant. Remember from the previous chapter that local L1 stall does not include L1 hits, it is stall time due to version upgrades or creating a new versions of a cache line for a thread that does not yet have a copy. This overhead partly replaces the overhead due to remote L1 stall seen in the CMP models. Neuralnet and Gzip also suffer from a larger fraction of L2 misses. They are also two of the applications which benefit the most from a larger L1 cache in the SMT simulations, that is the results shown in Figure 9.2.

In summary, the lower thread management overhead and reduction in remote L1 stall originating from thread communication makes SMT more efficient for thread-level speculation if the total issue width is the same in the two machines. However, for some applications the performance gain can be limited unless the SMT has a large enough L1 cache.

9.2.2 SMT and Run-Length Prediction

In the previous section none of the overhead reduction techniques were employed. Figure 9.5 shows the 4-thread simulations with run-length prediction and a threshold of 200. Results for two threads are similar but less pronounced.

It is clearly visible that run-length prediction does not improve performance as well for the SMT machine. Dblue and Neuralnet gain in speedup, but for several of

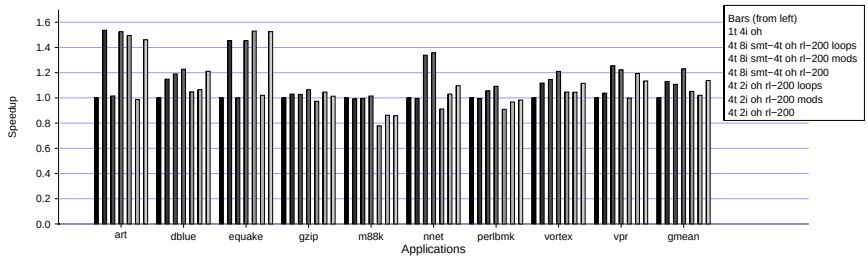


Figure 9.5: *TLS on an SMT with run-length prediction: Speedup on a 4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*

the other the speedup remains the same or even drops. This is not totally unexpected. Since the overhead is smaller for the SMT machine, it suffers less from spawning short threads. If run-length prediction is to be used with an SMT machine, the best threshold should most likely be smaller than for the CMP.

9.2.3 Thread Priority

In a multiprogrammed SMT workload the most important concern is typically to maximize throughput; the execution time for each thread is not the top priority. Therefore, most policies governing resource sharing in an SMT tries to maximize throughput.

However, for a TLS workload it is more important to make sure the non-speculative thread executes as quickly as possible for two reasons. First, the non-speculative thread is the only thread certain to make forward progress, other threads may fall prey to a misspeculation. Therefore, prioritizing the non-speculative thread is a way to reduce the risk for slowdowns. Second, if the non-speculative thread executes faster, it will produce and forward its output values quicker, thereby reducing the probability of misspeculations in more speculative threads.

In fact, the results in the previous section use some modifications to stock SMT policies in order to take this into account. It is assumed the SMT can only fetch instructions from one thread per cycle. A common algorithm used to distribute the fetch cycles amount the threads is ICOUNT [TEL95]. This algorithm counts the instructions available in the front-end of the processor (decode, issue queues) for each thread and fetches instructions for the thread with the lowest sum. This way, threads that execute fast will get more fetch cycles and thereby throughput is improved.

I have used a modified ICOUNT policy which is related to a modification presented by Wallace el al. for *threaded multiple path execution* (TME) [WCT98]. In TME, the alternate path of hard-to-predict branches are executes in free thread contexts on an SMT. However, the most likely path should have the highest priority.

Therefore, several tweaks of the ICOUNT algorithm are evaluated; one alternative is to use path priority and confidence, and bump the ICOUNT value (i.e. giving it lower fetch priority) for lower priority and lower confidence paths).

My modified policy similarly multiplies the ICOUNT with thread priority, where the non-speculative thread has priority 1, the lest speculative thread priority 2 and so on. This means the higher priority threads are allowed to fetch more often and have more instructions in the processor front-end, reducing the risk that they stall due to a lack of instructions in the issue queues.

A second modification, not used in TME, is that the higher priority threads are also prioritized in the issue stage. The non-speculative thread always gets to issue all its ready instructions first, followed by the least speculative thread and so on. The most speculative thread may only issue instructions if there are still empty issue slots and execution units when all the other threads have issued all their ready instructions.

Figure 9.6 shows results for the priority modifications (bars 2-4 in each cluster) compared to regular ICOUNT (bars 5-7). For most applications, using the priority modifications does not have a significantly impact on speedup. Art and Neuralnet are exceptions, where the priority policy performs better.

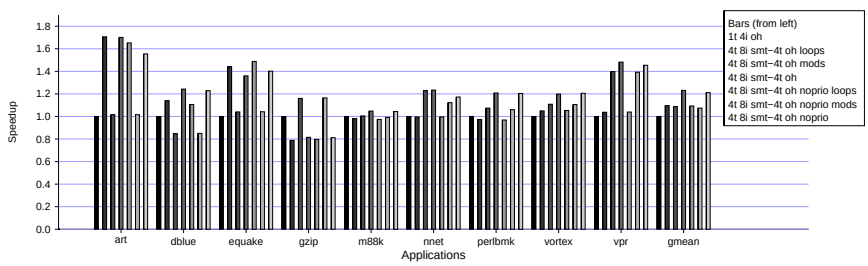


Figure 9.6: *TLS on an SMT with and without thread priority: Speedup on a 4-thread 8-issue SMT compared to a 4-way 2-issue CMP.*

For Equake, the result is even slightly better with regular ICOUNT, but running on the CMP is still the best option. Apparently, adjusting parameters governing the resource sharing does affect the result, which hints that the reason for the higher pipeline stall time and lower SMT speedup is connected to some form of interference between the threads in the SMT.

Looking back at Figure 9.3 we can see an effect of thread priority in the *Used/Exec* segments of the execution time breakdown. For several applications the CMP application have more used cycles; this is expected since the issue-width for the CMP cores is lower than the 4-issue sequential execution the result is normalized to. However, for the SMT threads the number of used cycles seems to be comparable to the

4-issue machine, despite the fact that four threads share the core. However, looking at for instance M88ksim or Vortex, one can see that the stall time is higher instead. This is because with the issue priority scheme the high priority threads run as if they had the whole machine available, i.e. a 4-issue machine. Low priority threads, on the contrary, experience more stall time. This does not always provide an advantage, however. In these cases the total running time for committed threads are on average about the same as on two 2-issue cores.

The priority was also extended to the L1 data cache, so that blocks from lower-priority threads were chosen for eviction before blocks from higher-priority threads when a block needed to be evicted from L1. This modification did not give any noticeable result. It would, in theory, help if there are many capacity misses, but this was not the case in my benchmark applications.

The bottom line is that normal ICOUNT works relatively well, but for a few applications it is better to prioritize the non-speculative and less speculative threads over more speculative threads.

9.3 TLS With a Single Speculative Thread

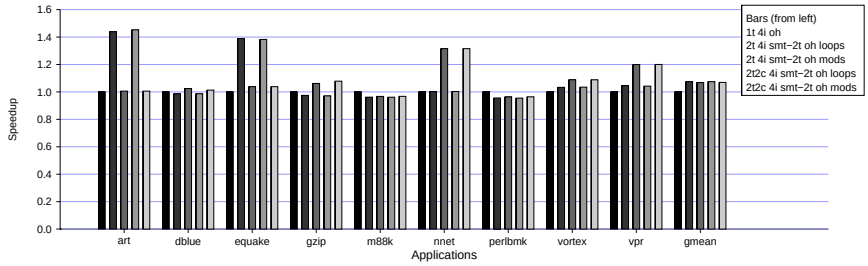
Supporting TLS with only a single speculative thread is less complex than with multiple threads. As a counterpoint to the flexible but relatively complex TLS implementation presented in Chapter 7, I will compare the results with the complex TLS model to the speedup it is possible to achieve with a machine using only one speculative thread and. In addition, I will discuss the hardware requirements for a possible implementation of such a machine.

9.3.1 Performance with a Single Speculative Thread

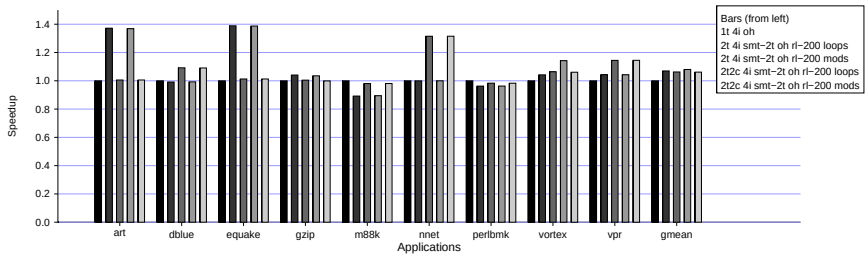
The performance of a machine with one speculative thread is evaluated both for an SMT, where both threads run on the same core, and on a 2-way CMP with one thread on each core.

The first results, in Figure 9.7, are for the SMT. The SMT is a 4-issue processor with support for two threads. For each application, there are results for loop- and module parallelism. Bars 2 and 3 in each cluster show results with the same parameters as in the previous section, i.e. there is an unlimited number of concurrent thread context, but only two running threads.

Bars 4 and 5 show results with only two thread contexts, i.e. if the speculative thread finishes before the non-speculative thread it will block the processor until the thread becomes head thread and can commit. With this policy, the machine only has to manage speculative state for one thread at a time, and the hardware can be simplified as described in Section 9.3.2.



(a) *2-thread SMT processor.*



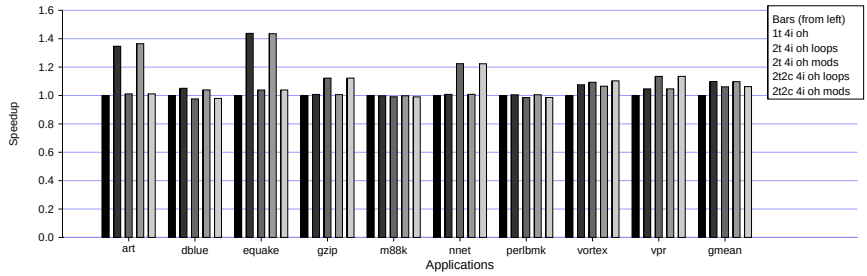
(b) *2-thread SMT processor with run-length prediction.*

Figure 9.7: *TLS with a single speculative thread on a 2-thread 4-issue SMT processor.*

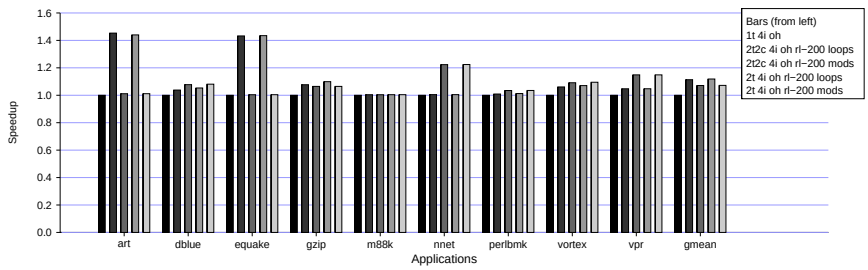
Figures 9.7(a) and 9.7(b) show results without and with run-length prediction respectively. As in the earlier SMT experiments, there is some loss of speedup for a few applications with run-length prediction, but the effect is quite small. The largest performance difference between supporting one and multiple thread contexts is for Vortex where the simpler scheme actually is faster for loops, due to somewhat fewer squashes. The speedup for this application is also improved due to successful loop unrolling. Clearly, with only two threads there is nothing to lose by only supporting speculative state for one speculative thread. The performance differences are very small.

Performance-wise, the speedup for Vpr is about half that of the best investigated machine (8-way CMP with run-length prediction, see Figure 8.16.) For Gzip, Neuralnet and Art more than half the speedup, and for Equake almost all the available speedup is still exploited with this simple machine.

Figure 9.8 shows the same experiments for a 2-way CMP, with 4-issue cores. The result is the same, additional thread contexts do not improve speedup when only two threads are used. The performance for the CMP machine is slightly lower than for the SMT, but even with this machine model much of the parallelism can be exploited



(a) *2-way CMP.*



(b) *2-thread CMP with run-length prediction.*

Figure 9.8: *TLS with a single speculative thread on a 2-way CMP with 4-issue cores.*

with one speculative thread.

These experiments have established two things. First, only one speculative thread is supported, no performance is lost by also supporting speculative state for only one thread, leaving the processor idle if there is load imbalance. Second, compared to results with the best 8-way CMP or 4-way SMT models, a reasonable amount of the available parallelism can be exploited with only one thread. With this in mind, it seems to be an interesting design point given the possible reductions in hardware complexity.

9.3.2 Hardware for a Single Speculative Thread

It is beyond the scope of this thesis to make thorough investigation of the most efficient support for TLS with only one speculative thread. However, one can easily enumerate some potential simplifications:

- There is no thread order to keep track of, an indicator showing which of the threads is speculative and which one is non-speculative is sufficient. Therefore, the elaborate TID scheme can be discarded in its entirety.

- The cache extensions do not need to record TID or version numbers, which reduces the cache overhead. However, the exposed load and store bits are still needed for dependence checking. Without TIDs and multiple speculative threads, there can be no mixing of speculative state from several threads in the same cache. Therefore, the scheme where commit and squash is indicated with bits in a thread list and the caches periodically sweeped to clean out old committed values is not needed. Instead, simpler commit and squash schemes like the gang commit or invalidate used in the speculative versioning cache can be used. Unlike SVC, there is no need for the next version pointer of version control logic, since there is only one speculative version.

- Dependence checking is simplified. Instead of comparing TIDs to make sure a write is from a less speculative thread, the only speculative thread knows that all writes from another thread is from the non-speculative thread.

- There are no scalability or pollution problems with shared prediction tables.

- If both threads run on the same SMT core, there is no need for a remote thread-start mechanism where register values are transferred over the on-chip interconnect. Instead, the initial values can be copied locally within the core. In addition, all speculative state can be managed in the L1 cache, and the memory hierarchy beyond the L1 is not TLS-aware.

In-core thread starts are described by prior TLS proposals for SMT processors, DMT [AD98], IMT, [PV03] and Marcuello and González [MG99b]. The last mentioned architecture propose storing speculative state in a modified L1, while DMT and IMT does not even allow speculative values to reach the level one cache; instead, memory accesses are held in the load/store queues until the thread is non-speculative. However, this only works if the speculative threads are very short, or the load/store queues exceptionally large.

9.4 Related Work

Previous TLS proposals for SMT processors have already been mentioned. DMT [AD98] and IMT [PV03] both manage the speculative threads completely within the SMT core. Therefore, thread sized should typically be small. The SMT architecture presented by Marcuello and González [MG99b] is specialized on loop

parallelization and lets several threads share the fetch bandwidth when several loop iterations execute along the same control path. However, none of these works explicitly compare the the advantages and disadvantages of an SMT architecture with a comparable CMP architecture.

As mentioned, Wallace et al. [WCT98] investigated different fetch policies for thread prioritizing in an SMT processor. Other SMT fetch and issue policies typically aim at maximizing throughput without regard for performance of the individual threads.

9.5 Conclusions and Future Work

In this chapter, I have investigated the performance of TLS on SMT processors. It was found that in general, TLS performs better on an SMT given equal fetch and issue capacity as a CMP. This is due to lower thread management overheads and reduced inter-thread communication costs. For some applications, however, it is necessary to scale the level one cache with the number of threads in order to achieve optimal performance on the SMT. Fetch and issue priority for less speculative threads over more speculative threads was also found to be effective for some applications.

On a chip with multiple SMT capable processors and integrated TLS support, a new set of questions beg to be answered. The resulting architecture is a chip with many available hardware threads, but they are not all equal in terms of available resources and thread management overhead. The effect of some of these differences have been demonstrated in this chapter. This presents a problem when scheduling the available speculative threads. When a new speculative thread is spawned, the speculation system should make a decision on which core to start the new thread. The best decision may well depend on a number of parameters related to both existing threads and the potential new thread; parameters such as cache and execution unit utilization, inter-thread communication requirements, and available ILP may affect this decision. Interesting future work would be to investigate scheduling policies in such an architecture.

The experiments with a single speculative thread showed that, somewhat surprisingly, most of the parallelism that could be exploited with the model presented in Chapter 7 could also be exploited with this simple model.

However, one should not forget that the sacrifice of adopting an implementation with a single speculative thread is scalability. Even with further improvements in the TLS model, for instance with compiler support or better scheduling policies, the performance potential is limited to a speedup of two in the ideal case. Taking inefficiencies which are hard to completely avoid into account, such as thread management overhead and some load imbalance, the realistic performance ceiling is further reduced.

10
Reflections and Outlook

The time has come to sum up my experiences with thread-level speculation. As tradition dictates, I will start with the bad news and finish on a more positive note.

When surveying the TLS literature, it is clear that the best results have been achieved with regular scientific applications such as the SPEC CPU floating point applications or the Olden benchmarks. The reported results for general integer applications varies, but the general theme is that while some performance gains are possible, the exploitable parallelism does not scale very well.

In my opinion, the fundamental drawback with attempting to extract parallelism from sequential binaries (or even from the source code level) is that by then most of the potential parallelism is unavailable for exploitation due to several steps of information loss. Trying to reverse-engineer the parallelism from the original problem expressed by a piece of software becomes increasingly difficult after each step of transformation.

First, a specific problem is converted into an algorithm. Typically this means one algorithm or solution out of many possible is chosen, possibly or even likely not the one which would be most amenable to parallelization if this is not a goal in the program construction phase. Then the algorithm is implemented in a high-level language, typically intended for sequential execution. Algorithms that could have been parallel are implemented with constructs made for sequential execution. Parallelizing compilers work at this level, trying to recover some of the parallelism

inherent in the problem. Third, the high-level language is transformed into machine code by a compiler. Even more information about the algorithms and data structures is lost when mapping the high-level language constructs into a sequence of simple instructions. This makes parallelization at run-time even more difficult.

Compilers can extract some parallelism even from sequential code, a typical example is a do-all loop where the iterations can be transformed into independent threads of execution. But much information is already lost, for instance if alternative data structures could be used or the order of computation changed.

In the approach taken in this thesis, i.e. trying to parallelize sequential binaries at run-time, the head-room for analysis is even smaller – since all analysis that is done will increase the run-time instead of reducing it, any time-consuming form of analysis must be ruled out. In short, the conditions for exploiting parallelism in this manner are not ideal. Compared to a compiler, run-time techniques have less information about the structure of the application. However, to some extent this is compensated with run-time information not available to the compiler. For instance, the run-length and misspeculation prediction techniques take advantage of such run-time information.

With this in mind, I find it promising to see that it is still possible to improve performance of some single-threaded applications with dynamic techniques. This can give an important performance boost for the vast library of existing applications. Even though only a limited amount of useful parallelism can be extracted from existing code, there is plenty of code around which makes it an important target.

However, I believe there are more reasons to pursue research in the fundamental concepts behind TLS. It has been demonstrated that TLS-like support can be used to speed up execution of critical sections (Martínez and Torrellas [MT02], Rajwar and Goodman [RG01], Rundberg and Stenstrom [RS03], and Sato et al. [SON00]), be used to for better software reliability (Oplinger and Lam [OL02]), as a debugging tool (Prvulovic and Torrellas [PT03]), and to aid parallel programming (Prabhu and Olukotun [PO03] and Hammond et al. [HCW+04]). Furthermore, I believe there is still room to improve the performance of TLS with compilers or binary translation tools. Recent work on such tools show promising results [DC04, OTKM05]. Finally, TLS support has similarities, and may be possible to combine with, runahead execution [DM97, MSWP03].

In other words, the support is not limited only to the topic of this thesis, that is run-time parallelization of existing binaries. The basic mechanisms of dependence checking, storing speculative state, and efficient thread-spawn mechanisms have many potential uses.

In my opinion, the most promising use is to leverage TLS for developing a simpler and lower-overhead parallel programming methodology. The traditional method of parallel programming involves thread libraries or message passing constructs, and operating system managed threads. This is useful for relatively easily parallelizable

applications, where sizable chunks of work can be divided into threads. With more available hardware threads and lower communication latency than previous multiprocessors, the chip multiprocessors potentially enable a larger class of applications to make use of multiprocessing. However, as I've showed in this thesis, this kind of fine-grained parallelism is more difficult to exploit and highly sensitive to the parallelization and communication overhead. Therefore, threads scheduled by the operating system and created using thread libraries will likely impose too much overhead to exploit the potential parallelism in many applications. With hardware support for threads that overhead can be radically reduced.

It seems clear to me that both traditional parallel programming, even with lightweight hardware threads, or automatic parallelization with TLS will face problems. The former because the burden on the programmer to handle communication, avoid deadlocks, live-locks and starvation, orchestrating for load balancing etc will be heavy in the face of numerous small threads. The latter because of the information loss problem preventing automatic parallelization from achieving large performance benefits in the general case. With TLS-like support acting as a safety-net against dependence violations, parallelization can potentially be more aggressive and place less burden on the programmer. This, however, would require new programming and profiling tools making good use of the capability. Projects like TCC [HWC+04, HCW+04] are showing the way towards such a methodology.

One might think that such a methodology may make run-time methods, such as the ones presented in this thesis, obsolete. However, I do not think this is necessarily the case. Aiding the programmers to create parallel programs with a simpler methodology only solves one part of the performance problem. Issues such as communication latencies, locality, and load balancing still remains to be addressed. Making the best scheduling decisions is a problem that I believe will often be best solved, at least in part, by run-time techniques. One reason is to make the code itself portable. When moving an application from one processor to another, some of the parameters affecting the scheduling decision will change. The number of available threads as well as the performance of each core, the memory and communication latencies and bandwidth are some parameters that will collude to change the optimal scheduling. Even within the same system, the current workload will affect the optimal scheduling.

Therefore, in order to make it possible for fine-grained parallel applications to adapt to changing environments without recompilation, dynamic thread scheduling is needed. Techniques based on dynamic performance monitoring and various forms of prediction, such as the ones presented in this thesis, could be very useful to guide a run-time system making such scheduling decisions.

Bibliography

[AAK+05] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, San Franscisco, California, February 2005.

[AD98] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 226–236. IEEE Computer Society, December 1998.

[AMD02] AMD Inc. *AMD Athlon Processor x86 Code Optimization Guide*, pages 235–242. AMD Inc., 2002.

[BDE+96] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoelflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12), 1996.

[CCYT05] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, 2005.

[CL03] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '03)*, pages 13–24. ACM Press, 2003.

[CL05] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems (TPDS).*, 16(6):562–576, 2005.

[CMT00] M. Cintra, J. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 13–24. ACM Press, June 2000.

[CO98] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 176–184. IEEE Computer Society, October 1998.

[CO03] M. K. Chen and K. Olukotun. The jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th Annual International Symposium on*

Computer Architecture (ISCA '03), pages 434–446. IEEE Computer Society, June 2003.

[CSG99] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Sofrware Approach*, pages 269–452. Morgan Kauffman, 1999.

[CSK+99] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (ssmt). In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society.

[CSL03] Y. Chen, R. Sendag, and D. J. Lilja. Using incorrect speculation to prefetch data in a concurrent multithreaded processor. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE Computer Society, April 2003.

[CT02] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the Eight International Symposium on High-Performance Computer Architecture (HPCA '02)*, pages 43–54. IEEE Computer Society, February 2002.

[CW99a] L. Codrescu and D. S. Wills. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 International Conference on Computer Design (ICCD '99)*, pages 428–435. IEEE Computer Society, October 1999.

[CW99b] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 40–46. IEEE Computer Society, October 1999.

[CWT+01] 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. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01)*, pages 14–25, New York, NY, USA, 2001. ACM Press.

[DC04] J. Dou and M. Cintra. Compiler estimation of load imbalance overhead in speculative parallelization. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 203–214, Washington, DC, USA, 2004. IEEE Computer Society.

[DLL+04] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*, pages 71–81, New York, NY, USA, 2004. ACM Press.

[DM97] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th ACM International Conference on Supercomputing (ICS '97)*, July 1997.

[DOO+95] P.K. Dubey, K. O'Brien, K.M. O'Brien, , and C. Barton. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '95)*, pages 109–121. IEEE Computer Society, June 1995.

[EWN04] M. Ekman, F. Warg, and J. Nilsson. An in-depth look at computer performance growth. Technical Report 04-9, Department of Computer Science and Engineering, Chalmers University of Technology, 2004.

[FS92] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 58–67. IEEE Computer Society, May 1992.

[FS96] M. Franklin and G. Sohi. Arb: A hardware mechanism for dynamic memory disambiguation. In *IEEE Transactions on Computers Vol. 45 No. 5*, pages 552–571. IEEE Computer Society, May 1996.

[GPL+03] M. J. Garzáran, M. Prvulovic, J. Llabería, V. Vinals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, February 2003.

[GPV+03] M. J. Garzáran, M. Prvulovic, V. Vinals, J. Llabería, L. Rauchwerger, and J. Torrellas. Using software logging to support multi-version buffering in thread-level speculation. In *Proceedings of the 2003 International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*. IEEE Computer Society, September 2003.

[GVSS98] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 195–206. IEEE Computer Society, February 1998.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12), 1996.

[HBJ02] S. Hu, R. Bhargava, and L. Kurian John. The role of return value prediction in exploiting speculative method-level parallelism. Technical Report TR-020822-02, University of Texas at Austin, August 2002.

[HCW+04] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS '04)*, pages 1–13. ACM Press, 2004.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[HP02] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, third edition, 2002.

[HWC+04] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA '04)*, pages 102–113. IEEE, June 2004.

[HWO98] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII '98)*, pages 58–69. ACM Press, October 1998.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[KL98] I. K. Kazi and D. J. Lilja. Coarse-grained speculative execution in shared-memory multiprocessors. In *Proceedings of the 1998 International Conference on Supercomputing (ICS '98)*, pages 93–100. ACM Press, July 1998.

[KM03] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[Kni86] Tom Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 105–112. ACM Press, 1986.

[KST04] J. Kalla, B. Sinharoy, and J. Tendler. IBM power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[KT98] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the International Conference on Supercomputing (ICS '98)*, July 1998.

[KT99] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[LL00] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, Vancouver, BC, June 2000.

[LS96] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 226–237. IEEE Computer Society, December 1996.

[LTS+06] W. Liu, J. Tuck, L. Seze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: A tls compiler taht exploits program structure. In *ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming (PPoPP '06). ACM Press, March 2006.

[LTW+96] Z. Li, J.-Y. Tsai, X. Wang, P.-C. Yew, and B. Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In *Languages and Compilers for Parallel Computing*, pages 175–191, 1996.

[Luk01] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01)*, pages 40–51, New York, NY, USA, 2001. ACM Press.

[LW92] M. Lam and R. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 46–57. IEEE, May 1992.

[LWS96] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII '96)*, pages 138–147. ACM Press, October 1996.

[MBM+06] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Conference on High Performance Computer Architecture (HPCA '06)*, February 2006.

[MCC+05] A. McDonald, JW. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2005.

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation plattform. *IEEE Computer*, pages 50–58, February 2002.

[MG99a] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 365–372. ACM Press, June 1999.

[MG99b] P. Marcuello and A. González. Exploiting speculative thread-level parallelism on a SMT processor. In *Proceedings of the International Conference on High Performance Computing and Networking (HPCN '99)*, pages 754–763, April 1999.

[MG00] P. Marcuello and A. González. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 595–604. IEEE Computer Society, May 2000.

[MG02] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*, page 55, Washington, DC, USA, 2002. IEEE Computer Society.

[MGT98] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 1998 International Conference on Supercomputing (ICS '98)*, pages 77–84. ACM Press, July 1998.

[MLM+98] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. Stenström, and H. Grahn. SimICS/sun4m: A virtual workstation. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 119–130. USENIX Association, June 1998.

[MS97] A. Moshovos and G. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97)*. IEEE, May 1997.

[MSWP03] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA '03)*, pages 129–140, February 2003.

[MT02] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS '02)*, pages 18–29. ACM Press, Oct 2002.

[MTG99] P. Marcuello, J. Tubella, and A. González. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO '99)*, pages 230–237. IEEE Computer Society, December 1999.

[OHL+97] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, 1997.

[OHL99] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 303–313. IEEE Computer Society, October 1999.

[OHW99] K. Olukotun, L. Hammond, and M. Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of the 1999 International Conference on Supercomputing (ICS '99)*, pages 21–30. ACM Press, June 1999.

[OKP+01] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *International Conference on Supercomputing (ICS '01)*, pages 368–380, June 2001.

[OL02] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 184–196. ACM Press, October 2002.

[ONH+96] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*, pages 2–11. ACM Press, October 1996.

[OTKM05] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Washington, DC, USA, 2005. IEEE Computer Society.

[PGRT01] M. Prvulovic, M. J. Garzzáran, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, pages 204–215. IEEE Computer Society, July 2001.

[PGTM99] M. A. Postiff, D. A. Greene, G. S. Thyson, and T. N. Mudge. The limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News*, 217(1):31–34, March 1999.

[PO03] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pages 1–12. ACM Press, June 2003.

[PT03] Milos Prvulovic and Josep Torrellas. Reenact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA '03)*, pages 110–121, New York, NY, USA, 2003. ACM Press.

[PV03] I. Park and T. N. Vijaykumar. Implicitly-multithreaded processors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pages 39–50. IEEE Computer Society, June 2003.

[QMS+05] C. G. Quinones, C. Madriles, J. Sánches, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 269–279. ACM Press, June 2005.

[RG01] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO '01)*, pages 294–305. IEEE Computer Society, December 2001.

[RG03] R. Rajwar and J. R. Goodman. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125, Nov-Dec 2003.

[RHL05] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[RJSS97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 138–148. IEEE Computer Society, December 1997.

[RP95] L. Rauchwerger and D. A. Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*, pages 218–232, 1995.

[RP99] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):160–180, 1999.

[RS01] P. Rundberg and P. Stenstrom. An all-software thread-level data dependence speculation system for multiprocessors. *The Journal of Instruction-Level Parallelism*, 3, October 2001.

[RS03] P. Rundberg and P. Stenstrom. Speculative lock reordering: Optimistic out-of-order execution of critical sections. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE Computer Society, April 2003.

[RSC+05] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a cmp can be energy efficient. In *Proceedings of the International Conference on Supercomputing (ICS '05)*. ACM, June 2005.

[RTL+05] 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. In *Proceedings of the International Conference on Supercomputing (ICS '05)*. ACM, June 2005.

[SBV95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425. ACM Press, June 1995.

[SCM97] J. G. Steffan, C. B. Colohan, and T. C. Mowry. Architectural support for thread-level data speculation. Technical Report CMU-CS-97-188, Carnegie Mellon University, November 1997.

[SCZM00] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 1–12. ACM Press, June 2000.

[SCZM02] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the Eight International Symposium on High-Performance Computer Architecture (HPCA '02)*. IEEE Computer Society, February 2002.

[SD98] Y. Song and M. Dubois. Assisted execution. Technical Report CENG 98-25, Department of EE-Systems, University of Southern California, October 1998.

[SM98] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 2–13. IEEE Computer Society, February 1998.

[SON00] T. Sato, K. Ohno, and H. Nakashima. A mechanism for speculative memory accesses following synchronizing operations. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, page 145. IEEE Computer Society, May 2000.

[SPHC02] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 45–57. ACM Press, October 2002.

[SS97] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 248–258. IEEE Computer Society, December 1997.

[Ste90] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, 1990.

[TEL95] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA '95)*, pages 392–403. ACM Press, June 1995.

[TG98] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 14–23. IEEE Computer Society, January 1998.

[TJY99] J.-Y. Tsai, Z. Jiang, and P.-C. Yew. Compiler techniques for the superthreaded architectures. *International Journal of Parallel Programming*, 27(1):1–19, 1999.

[Tre99] M. Tremblay. Majc: Microprocessor architecture for java computing. In *HotChips '99*, August 1999.

[TY96] J.-Y. Tsai and P.-C. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46. IEEE Computer Society, October 1996.

[VS98] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 81–92. IEEE Computer Society, December 1998.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV '91)*, pages 176–189. ACM Press, April 1991.

[WCT98] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*, pages 238–249. ACM Press, June 1998.

[WS01] F. Warg and P. Stenstrom. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*, pages 221–230. IEEE Computer Society, September 2001.

[WS03] F. Warg and P. Stenstrom. Improving speculative thread-level parallelism through module run-length prediction. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '03)*. IEEE Computer Society, April 2003.

[WS05] F. Warg and P. Stenstrom. Reducing misspeculation overhead for module-level speculative execution. In *Proceedings of the 2005 International Conference on Computing Frontiers (CF '05)*. ACM Press, May 2005.

[WWFH03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA '03)*, pages 84–97. ACM Press, 2003.

[ZCSM02] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 171–183. ACM Press, October 2002.

[ZCSM04] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the international symposium on Code generation and optimization (CGO '04)*, page 39, Washington, DC, USA, 2004. IEEE Computer Society.

[ZF03] M. Zahran and M. Franklin. Dynamic thread resizing for speculative multithreaded processors. In *Proceedings of the International Conference on Computer Design (ICCD '03)*, October 2003.

[ZRT98] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, pages 162–173. IEEE Computer Society, Jan 1998.

[ZRT99] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA '99)*, pages 135–. IEEE Computer Society, Jan 1999.

[ZUR04] G. Zhang, P. Unnikrishnan, and J. Ren. Experiments with auto-parallelizing spec2000fp benchmarks. In *Proceedings of 17th International Workshop on Languages and Compilers for High Performance Computing (LCPC '04)*, volume 3602 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 2004.

List of Figures

List of Tables