Exploring Efficient Architecture Design for Thread-Level Speculation — Power and Performance Perspectives

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Venkatesan Packirisamy

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor Of Philosophy

June, 2009

ABSTRACT

With the advent of multi-threaded (e.g. simultaneous multi-threading (SMT) [1, 2]) and/or multi-core (e.g. chip multiprocessors (CMP) [3, 4]) architectures, now the challenge is to utilize these architectures to improve performance of general-purpose applications. However, traditional parallelizing compilers often fail to effectively parallelize general-purpose applications which typically have complex control flow and excessive pointer usage. Thread-Level Speculation (TLS) have been proposed to simplify the task of parallelization by using speculative threads. Though the performance of TLS has been studied in the past, its power consumption, power efficiency and thermal behavior are not well understood. Also previous work on TLS have concentrated on multi-core based architectures and relatively little has been done on supporting TLS on multi-threaded architectures. With increasing multi-threaded/multi-core design choices, it is important to understand the benefits of the different types of architectures.

The goal of this dissertation is to explore architecture techniques to efficiently implement TLS in future multi-threaded/multi-core processors. The dissertation proposes a novel cache-based architecture to support TLS in multi-threaded SMT architecture. A detailed study on the efficiency of different TLS architectures was conducted by comparing their performance, power and thermal characteristics. To improve efficiency, the dissertation proposes a novel SMT-CMP based *heterogeneous* architecture which combines the advantages of both SMT and CMP architectures. The dissertation also proposes novel architecture and compiler techniques to efficiently extract speculative parallelism from multiple loop levels.

Acknowledgements

First I would like to thank my advisors Prof. Pen-Chung Yew and Prof. Antonia Zhai for their guidance and support during my Ph.D. Without their constant encouragement and their patience with me I would not have survived the Ph.D. process.

I would also like to thank my thesis committee members - Prof. Wei-Chung Hsu, Prof. David Lilja and Prof. Zhi-li Zhang for providing guidance and suggestions at various stages of my Ph.D.

Also, I am indebted to my colleagues: Jin Lin, Tong Chen, Shengyue Wang, Xiarou Dai, Jinpyo Kim, Kiran S. Yellajyosula, Guojin He, Yangchun Luo, Jin-Woo Jung, Wei-Lung Hung, Pei-Hung Lin, Jagan Jayaraj, Guoqiang Yang, Yang Liu, and Harish Barathvajasankar for their help and encouragement. Also I would like to thank all my friends, especially Ranganathan Balasubramanian and Krishna Raghavendra Rao for their encouragement and for making my very long stay at Minneapolis quite fun.

Finally I would like to thank my parents and my sister for their support, encouragement and love that helped me through my Ph.D.

.

This thesis is dedicated to the memory of Kiran S. Yellajyosula

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuous clock rate improvement on microprocessors in the past three decades has stalled in early 2000's because of power and thermal considerations. It prompted computer industry to adopt multi-threaded (e.g. simultaneous multi-threading (SMT) [1], hyper-threading [2]), and/or multi-core (e.g. chip multiprocessors (CMP)) [3, 5] architectures in the hope of continuing the performance improvement without increasing the clock rate and its associated power and thermal problems. With the advent of multi-threaded and multi-core architectures, now the challenge is to utilize these architectures to improve performance of general-purpose applications. Automatic compiler parallelization techniques have been developed and found to be useful for many scientific applications that are floating-point intensive. However, when applied to general-purpose integer-intensive applications that have complex control flow and excessive pointer accesses, traditional parallelization techniques become quite ineffective, as they need to conservatively ensure program correctness by synchronizing all potential dependences in the

program. Another approach is to allow the programmer to explicitly create parallel threads and insert synchronizations. This approach is often error prone and puts a huge burden on the programmer.

There have been numerous studies on hardware support for speculative threads, which intend to ease the creation of parallel threads for programmers and compilers. Recently, Hardware Transactional Memory (HTM) has been proposed to aid the development of parallel programs; Thread-Level Speculation (TLS) has been used to exploit parallelism in sequential applications that are difficult to parallelize using traditional parallelization techniques. For example, a loop that contains an inter-thread data dependence due to loads and stores through pointers cannot be parallelized using traditional compilers; but with the help of TLS, the compiler can parallelize this loop speculatively and relying on the underlying hardware to detect and enforce inter-thread data dependences at run-time.

There has been a significant amount of research done on Thread Level Speculation (TLS) [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29] on how to automatically extract speculative parallelism from programs and on how to support TLS in hardware. From the previous work the performance behavior of multi-core (CMP) based TLS is well understood, but very little has been done to understand other important characteristics like power consumption, power efficiency and thermal behavior. Also the multi-threaded (SMT) based TLS has not been well understood and the current techniques rely on complex structures that can support only small speculative threads [30, 7].

With the current trend towards multi-threaded/multi-core architectures, the microprocessor

designer is now presented with a variety of multi-threaded/multi-core design choices. While comparative studies [31, 32, 33, 34, 35] on the advantages of CMP and SMT based architectures have been conducted using various workloads, their relative behavior in the case of TLS is not known. To efficiently implement TLS in future multi-threaded/multi-core processors it is important to understand the relative advantages of the various design choices when supporting TLS.

Also with the current trend towards increasing number of threads/cores supported in the microprocessor, it is important to utilize all the available threads/cores to extract performance in applications. To efficiently utilize the available threads/cores, it is important to extract parallelism from multiple loop and function levels. While most TLS techniques have studied only single-level parallelism, it is important to develop techniques to efficiently support multiple levels of TLS.

This dissertation addresses some of these issues. First we propose a novel cache-based architecture to support TLS in SMT processors. Then we perform a detailed trade-offs study by comparing the performance, power consumption and thermal behavior of SMT based TLS with the CMP based TLS architecture. Based on our understanding of the relative merits of SMT and CMP based TLS architecture, we propose a SMT-CMP based *heterogeneous* architecture and show its improved efficiency. We also propose compiler and architecture techniques to improve scalability of TLS by exploiting speculative threads from multiple loop levels.

1.1 Related work

Automatic parallelization techniques have been extensively studied in the past and found to be successful in parallelizing scientific programs [36, 37]. But when applied to general purpose programs that have ambiguous data dependences and complex control flow, the traditional compiler is forced to conservatively synchronize on all potential inter-thread dependences.

One way to overcome this limitation is to use Thread-Level Speculation(TLS) where the compiler can ignore these ambiguous or low-probability dependences and speculatively parallelize the application. But such speculation requires hardware or software support to ensure correctness in case the inter-thread dependences do occur at run-time. Software techniques [38, 39, 40] to support TLS suffer from large overhead, leading to only limited performance.

Architecture techniques to support TLS have been extensively studied in the past and found to be successful due to their lower overhead. Multiscalar [11] introduced the concept of hardware based TLS and initially used hardware buffers called Address Resolution Buffers(ARB) [41]. Later studies relied on shared memory based cache coherence protocols to support TLS [9, 13, 14, 15, 17, 18, 42].

Compared to CMP based TLS design, SMT based TLS has not been extensively studied. Current techniques use complex hardware buffers such as the existing Load-Store queues(LSQs) in the processor, which due to their smaller size can only support smaller threads.

While the performance aspect of the TLS architectures are well understood, their power efficiency has not been well understood. Renau *et. al* [43] compared the power consumption of CMP based TLS architecture with the superscalar based architecture and found the TLS to be

more power efficient. However other design choices such as SMT were not considered.

Power consumption, energy efficiency and thermal characteristics of SMT and CMP have been well understood under various workloads: On parallel programs [31] and mobile workloads [32], SMT processors outperform CMP processors. However, on multimedia workloads, CMP is more efficient [33]. In the context of multi-program workload, Li *et. al* [34] found that SMT is more efficient for memory-bound applications while CMP is more efficient for CPU-bound applications; Burns *et. al* [35] found that SMT can achieve a better single thread performance, but CMP can achieve a higher throughput.

Recently there are some studies that utilize a *heterogeneous* architecture to improve the energy efficiency. In most previous studies *heterogeneous* architectures [44, 45, 46], different cores with varying complexity (or varying frequency) are combined together to for a multi-core processor. The multi-program workloads are either statically or dynamically mapped to the various cores to improve the overall efficiency of the workload.

Most research on TLS concentrated on exploiting TLS on a single loop level or a single function call level. Renau *et. al* [47] proposed a hardware based technique to extract speculative threads from multiple levels. Here the threads are aggressively extracted from multiple levels and it relies on complex hardware based runtime system to efficiently utilize the available cores and filter out non-performing loop levels.

1.2 Dissertation contributions

In this dissertation, we make several contributions that facilitates efficient implementation of speculative threads in multi-threaded/multi-core processors:

1. The thesis analyzes the performance potential for TLS in the more recent SPEC 2006 benchmarks and shows the importance of supporting TLS. By comparing the performance of SPEC 2006 with older SPEC 2000 benchmarks, the dissertation shows a trend towards more parallel applications which need TLS support.

2. The thesis proposes a cache-based architecture to support TLS in SMT processors. It shows that the new technique proposed can outperform other known methods to support TLS in SMT.

3. The thesis performs a detailed comparison of performance, power and thermal characteristics of both SMT based and CMP based TLS architectures. It is shown that depending on the individual benchmark behavior, different architectures show better efficiency.

4. To exploit the advantages of both SMT and CMP based TLS, the thesis proposes a novel SMT-CMP based *heterogeneous* architecture to efficiently support TLS. The thesis shows that this novel architecture can lead to better efficiency than the homogeneous SMT or CMP based TLS architecture.

5. The dissertation proposes a compiler based approach to extract TLS parallelism at multiple loop levels. The dissertation shows how the architecture design can be simplified

with appropriate compiler support.

The rest of the dissertation is organized as follows:

- Chapter 2 describes the evaluation framework used in this thesis.
- Chapter 3 discusses the potential for TLS performance in SPEC 2006 and compares it with the performance of SPEC 2000.
- Chapter 4 presents a novel cache-based architecture to support TLS in SMT processors.
- Chapter 5 presents a detailed comparison of SMT based TLS architecture with CMP based TLS architecture.
- Chapter 6 shows the potential gain in efficiency by using a heterogeneous multi-core architecture.
- Chapter 7 presents the architectural and compiler techniques to exploit multi-level TLS.
- Chapter 8 presents our conclusions and opportunities for future work.

Chapter 2

Evaluation Framework

In this chapter, we present the details on the TLS model used and a description of the our evaluation framework.

2.1 TLS execution model

Under thread-level speculation (TLS), the compiler partitions a program into speculatively parallel threads without having to decide at compile time whether they are independent. At runtime, the underlying hardware determines whether inter-thread data dependences are preserved, and re-executes any thread for which they are not. The most straightforward way to parallelize a loop is to execute multiple iterations of that loop in parallel. In our baseline execution model, the compiler ensures that two loops within a loop-nest will not be speculatively parallelized simultaneously. In Chapter 7, we will study the potential for supporting speculative threads at multiple nesting levels.

2.1.1 Elements of TLS:

Thread forking: A fork instruction is inserted by the compiler at the beginning of each iteration. On execution of the *fork* instruction, a new thread is created in the next available hardware context after a fixed delay.

Speculative buffering: When executing the speculative thread, all data created are speculative and should not be allowed to modify the non-speculative context of the application. To avoid this the hardware buffers all the results from the speculative thread.

Dependence checking: To enforce correctness of the application the hardware monitors for any dependency violations by the speculative thread. All memory addresses loaded by the speculative thread are tracked by the hardware and checked for dependency violations after every store from non-speculative thread (or other earlier speculative threads).

Thread squashing: If a violation is detected, the hardware restarts the thread and all the speculative data along with the special bits used to track speculation are discarded.

Synchronization: To communicate scalar values across threads, the compiler inserts special synchronization instructions *wait* and *signal*. Frequently occurring memory dependences are also synchronized as described in [48].

Thread committing: When the non-speculative thread finishes execution, the next immediate thread in fork order becomes the new non-speculative thread. When a thread becomes non-speculative, its speculative state is integrated with the non-speculative state of the

program. Also the special bits used to track dependences and buffer speculative results are discarded.

2.1.2 TLS hardware model:

In this thesis we use both SMT and CMP based architectures to support TLS. SMT support for TLS is described in detail in chapter 4.

For the CMP based TLS architecture, we use a cache-based protocol based on STAM-Pede [49]. When executing speculative threads, speculative stores will be buffered in the private L1 data cache and speculative loads are marked using a special bit in the cache. When a dependence violation is detected, the violating thread and all its successors are restarted. When the violating thread is squashed, all speculation marker bits in the L1 data cache are reset with a gang-clear (1 cycle). When a thread commits, it sends a signal to its immediate successor, and the latter becomes the new non-speculative thread. More details on the TLS architecture model used can be found in [49].

In this thesis, we allow memory based communication between speculative threads. So if a speculative thread does a store which finds latter versions of the same address in the other caches, the target cache line is 'combined' with the current value if the latter thread had not speculatively read from the same location. For example, say thread-3 writes to location A it creates a version in its private cache. Say now thread-2 writes to location A+1 and it finds a version of the cache line created by thread-3 in its private cache (assuming A and A+1 map to the same cache line). Since thread-3 has not read from location A+1, the store does not cause

speculation failure. So we allow thread-2's store to update the value in location A+1 in thread-3's cache line. Now if thread-3 issues a load to location A+1, it can get the latest value of A+1 stored by thread-2.

Frequently occurring memory-based dependences and register-based scalar dependences are synchronized by inserting special instructions as shown in [21, 20]. In this thesis, we assume a fast interconnection network to communicate values between the threads. (The values can also be communicated through the shared L2 cache.)

2.2 TLS compiler

Our compiler infrastructure is built on the Open64 3.0 Compiler [50], an industrial-strength open-source compiler targeting Intel's Itanium Processor Family (IPF). To create and optimize speculative parallel threads, the compiler must perform accurate performance trade-off analysis to determine whether the benefit of speculative parallel execution outweighs the cost of failed speculation and then aggressively optimize loops that benefit from speculation. As shown in Fig. 2.1, the compiler performs such analysis and optimizations based on loop nesting, edge, as well as data dependence profiling (using the train input sets). The TLS compiler has two distinct phases – loop selection and thread optimization:

Loop selection: In the loop selection phase, the compiler first estimates the parallel performance of each loop based on the cost of synchronizations, as well as the probability and the cost of speculation failures. The compiler then chooses to parallelize a set of loops

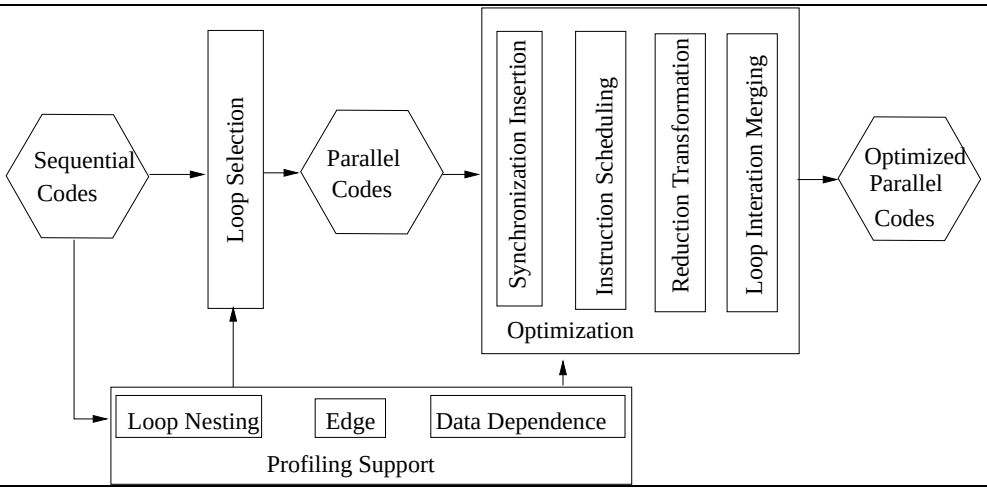


Figure 2.1: Compilation infrastructure

that maximize the overall program performance based on such estimations [25, 48].

Code Optimization: The selected parallel loops are optimized with various compiler optimization techniques to enhance TLS performance: (i) all register-resident values and memory-resident values that cause inter-thread data dependences with more than 20% probability are synchronized [21]; (ii) instructions are scheduled to reduce the critical forwarding path introduced by the synchronization [20, 48]; (iii) computation and usage of reduction-like variables are transformed to avoid speculation failures [48]; and (iv) consecutive loop iterations are merged to balance the workload of neighboring threads [48].

2.3 Simulator framework

We use a trace-driven simulator to simulate both the SMT and CMP architectures. Prior TLS research typically simulated the first billion instructions in each benchmark after skipping the initialization portion. Yi *et. al* [51] have shown that such truncated simulation does not cover all

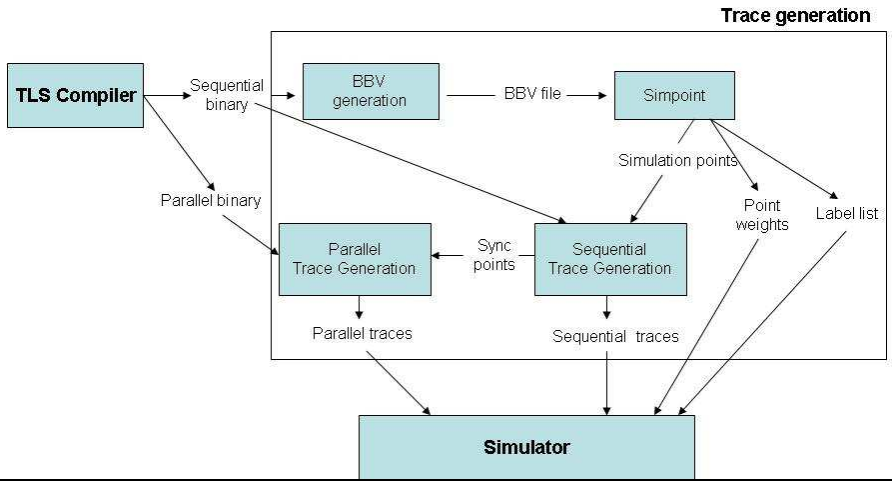


Figure 2.2: Trace Generation Framework

phases in a benchmark, and thus can potentially miss important program behavior that only appear in the later parts of the execution. To improve simulation accuracy and to reduce simulation time, we have adopted a SimPoint-based sampling technique [52].

2.3.1 Trace generation

Before generating traces for the simulator, *S*impoint samples have to be selected. The sequential binary created by the compiler is used to generate Basic Block Vectors (BBV) which are then used to select *S*impoint samples. The selected sample points are fed into the sequential trace generator which creates traces for corresponding points selected. The trace generator is based on the PIN instrumentation tool [53]. The tool instruments all instructions to extract information such as instruction address, registers used, memory address for memory instructions, opcode, etc. The collected information is written to the output trace file which will be used by the simulator.

The parallel trace generator is augmented so that it selects the exact same code regions in

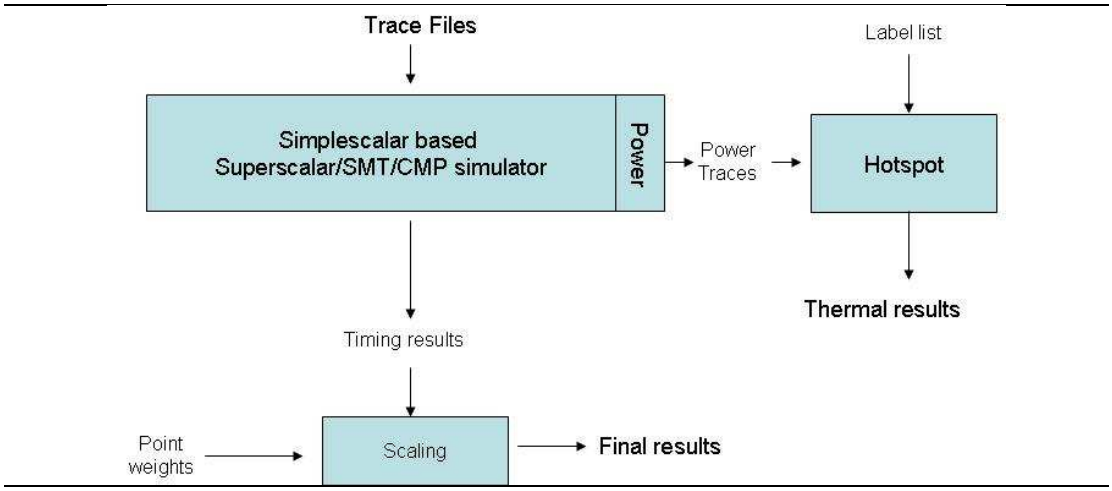


Figure 2.3: Simulator Framework

the parallel executables as that in the sequential samples. Fig. 2.2 gives a block diagram of the trace generation framework.

2.3.2 Simulation

We use a detailed out-of-order simulator based on Simplescalar [54] to simulate both SMT and CMP architectures. Fig. 2.3 shows a block diagram of the simulator framework.

We not only model register renaming, reorder buffer, branch prediction, instruction fetching, branch mis-prediction penalty and memory hierarchy, but also extend the infrastructure to model different aspects of TLS execution including explicit synchronization through signal/wait and cost of thread commit/squash.

To estimate power consumption of the processors, the simulator is integrated with the Wattch [55] power model. The power consumption for the common bus in the CMP architectures is simulated using Orion [56]. The power traces generated by the simulator are fed to HotSpot [57] to evaluate the thermal behavior of the system.

2.3.3 Thermal simulation

While sampling techniques described here can significantly reduce the simulation time without compromising much simulation accuracy for processor performance and power consumption, these techniques cannot be applied to thermal simulation. To accurately simulate the thermal effects, we must construct the power consumption trace for the execution of the entire benchmark. In our study, we do so with the power traces collected for the Simpoint samples.

Let's assume that the original full execution trace (without sampling) is broken into a set of m segments, $\{t_1, t_2, \ldots, t_m\}$; we refer to this list as T. In *Simpoint* sampling, a subset of T, say S, is selected for detailed simulation. For each selected segment, there is a corresponding power trace p. Let us refer to the set of such power traces as P. For every segment t_i in the original trace T, there is a corresponding segment say s_j in the samples S that belong to the same phase as t_i. The behavior of t_i is similar to that of s_j as they belong to the same phase. So we could approximate the power behavior of t_i by using the power trace of s_j in its place. Using this method, we construct the power trace of the entire execution sequence $\{t_1, t_2, \ldots, t_m\}$ by using the power trace of the corresponding sampled segment s_j for each t_i. The resulting *approximate* power trace is fed to HotSpot [57] to study the thermal behavior of the different configurations. We found that by taking advantage of the phase behavior, such thermal estimate is quite accurate.

2.3.4 Benchmarks

We use the SPEC 2000 and SPEC 2006 benchmarks to evaluate our techniques. All the benchmarks are run using the *ref* input set. For trace generation we generate a maximum of 10 traces (-maxK=10) per benchmark and each trace is 100 Million instructions in size. Table 2.1 gives the details about the loops selected and their execution time coverage. As shown in Table 2.1, some benchmarks like *art, mcf, equake, ammp, sphinx3, namd, lbm, libquantum, and hummer* the compiler was able to extract parallelism in more than 95% of the dynamic execution coverage. While in other benchmarks like *perlbmk, crafty, gap and gobmk* the compiler extracted parallelism from less than 30% of the dynamic execution coverage.

Table 2.1: Details of Benchmarks

SPEC 2000 Benchmarks				
Benchmark	Total no. of loops	No. of dynamic loop-nesting levels	No. of loops selected	Dynamic coverage of selected regions
perlbmk	751	8	9	25%
art	74	6	25	99%
vpr_place	416	5	3	55%
gcc	2429	10	98	83%
parser	537	10	40	82%
vpr_route	416	6	19	94%
mcf	53	5	13	98%
equake	91	5	9	93%
ammp	358	10	21	99%
twolf	888	7	20	47%
bzip2	159	9	19	81%
mesa	903	6	3	63%
gzip	191	6	6	99%
crafty	405	9	3	13%
vortex	230	7	8	67%
gap	1659	10	8	30%
SPEC 2006 Benchmarks				
bzip2	232	11	14	46%
mcf	52	5	6	97%
gobmk	1265	22	13	21%
sphinx3	609	8	21	97%
namd	619	4	50	99%
povray	1311	15	5	63%
astar	116	6	7	73%
lbm	23	3	2	99%
h264ref	1870	15	36	79%
libquantum	94	4	5	99%
sjeng	254	10	6	40%
hmmer	851	5	5	96%
milc	421	11	22	85%

Chapter 3

Benchmark Analysis

Before we discuss architecture features that enhances efficiency of TLS, it is important to understand the potential for TLS. In this chapter we present an analysis of benchmark behaviors.

Though TLS has been extensively studied in the past, it is not clear how much TLS could benefit more recent benchmarks such as SPEC 2006 [58], which represent a different class of applications. Some recent studies [59] on SPEC 2006 benchmarks have shown limited potential for TLS (less than 1%) under very conservative assumptions. In this chapter, we re-examine some of these issues and provide a more realistic assessment of TLS on this benchmark suite using our Open64 based TLS compiler developed by our group [48]. Furthermore, we compare the behavior of SPEC 2000 and SPEC 2006 benchmarks to show that there is more potential parallelism for TLS in SPEC 2006 than in SPEC 2000.

To the best of our knowledge the only study on the TLS performance on SPEC 2006, Kejariwal *et. al* [59] did not take into account the effect of compiler optimizations that could

improve the performance of TLS, despite previous studies [26, 20, 21, 48] have shown that they can significantly improve the efficiency of TLS. Furthermore, Kejariwal *et. al* [59] only considered innermost loops for TLS, which is also known to be a limiting factor when extracting TLS. In this chapter, our study is not limited to innermost or outermost loop level, rather we attempt to parallelize all loops that can potentially benefit from TLS. More importantly, instead of a high-level study on performance potential of TLS, we use a state-of-the-art TLS compiler to parallelize TLS loops and study their performance using a detailed simulation infrastructure. Our results show that, with TLS-oriented compiler optimizations and accurate selection of loops, we could achieve an geometric mean of about 60% speedup with four cores for SPEC 2006 benchmarks over what could be achieved by a traditional parallelizing compiler such as Intel's ICC compiler. In comparison, the SPEC 2000 benchmarks achieve only about 32% geometric mean speedup.

3.1 Related work

There has been a large body of research work on architectural design and compiler techniques for TLS [18, 49, 20, 21, 22, 26], but these work are all based on SPEC 2000 [60] or other older benchmarks. The SPEC 2006 benchmarks [58] represent a newer class of applications and it is important to examine whether the conclusions drawn for SPEC 2000 will hold for these applications. In this chapter, we address this issue by conducting a detailed study of SPEC 2006 benchmarks using a state-of-the-art TLS compiler [48].

Several limit studies have examined the performance potential of TLS in older benchmarks.

Oplinger *et. al* [16] presented a study on the limits of TLS performance on some SPECint95 benchmarks. The impact of compiler optimizations and the TLS overhead were not taken into account in that study. Similarly, Warg *et. al* [61] presented a limit study for module-level parallelism in object-oriented programs. In contrast, in this study, our aim is to illustrate the realizable performance of TLS using a state-of-the-art TLS compiler, while taking into account various TLS overheads.

Kejariwal *et. al* [62] separated the speedup achievable through traditional thread-level parallelism from that of TLS using the SPEC2000 benchmarks assuming an *oracle* TLS mechanism. They [59] later extended their study to the SPEC 2006 benchmarks. It is worth pointing out that they concentrated on only inner-most loops and used probabilistic analysis to predict TLS performance. We also separate the speedup achievable through traditional non-speculative compilation techniques from that requires TLS support; however, we consider all loop-levels instead of just the inner-most or the outer-most loops. Furthermore, they manually intervened to force the compiler to parallelize loops that were not automatically parallelized due to ambiguous dependences. In this chapter, we utilize an automatic parallelizing compiler that performs trade-off analysis using profiling information to identify parallel threads—no programmer intervention needed.

3.2 Data dependence analysis of SPEC 2006 loops

Consider the example loop shown in Figure 3.1(a) with two cross-iteration dependences: a register-based dependence through register $r2$ and a potential memory-based dependence through

pointer p and q. In each iteration of the loop, the value of $r2$ from the previous iteration is required, thus the compiler must insert synchronization operations (the `wait/signal` pair) to ensure correct execution (shown in Figure 3.1(b)). In the case of the memory-based dependence, the cross-iteration dependence only occurs when the load through pointer p accesses the same memory location as the store through pointer q from a previous iteration. Since the compiler is unable to determine the address pointed to by p and q at compile time, it must insert synchronization operations (the `wait_mem/signal_mem` pair) as shown Figure 3.1(b). However, such synchronization can potentially serialize execution unnecessarily, as shown in Figure 3.1(c). With the help of TLS, the compiler can parallelize this loop by ignoring ambiguous data dependences and relying on the underlying hardware to detect and enforce all data dependences to ensure correctness at runtime. Figure 3.1(d) shows the loop executing in TLS mode: when the store through pointer q in $thread1$ accesses the same memory location as the load through pointer p in $thread3$, the hardware detects the dependence violation and restarts the violating thread. $Thread2$, which does not contain the destination of any inter-thread data dependence, is able to execute in parallel with $thread1$. This parallelism cannot be exploited without the help of TLS. However, if the dependence between $store*q$ and $load*p$ occurs frequently causing speculation to fail often, it can potentially degrade performance. In such cases, it is desirable for the compiler to insert explicit synchronization to avoid mis-speculation penalty.

Understanding the inter-thread data dependence patterns in an application is critical for estimating its TLS performance potential. In this section, we analyze the dependence information

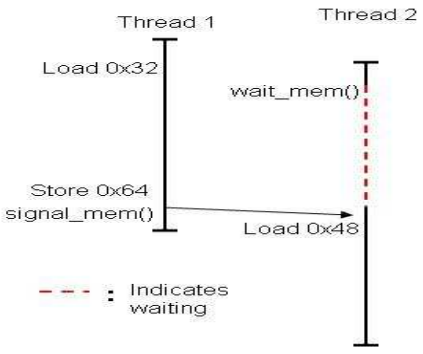
collected through data dependence profiling, and estimate the importance of TLS hardware support in exploiting parallelism in the SPEC 2006 benchmarks.

```
do  {
   ...
   load *p;
   ...
   r3 = r2 + 2;
   ...
   r2 = r1 + 1;
   ...
   store *q;
} while (condition)
```

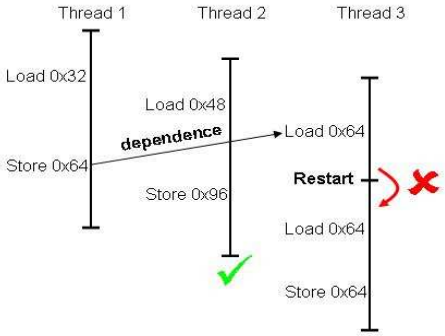
(a) A loop with loop-carried register-based and memory-based data dependences.

```
do  {
   ...
   wait_mem()
   load *p;
   ...
   wait()
   r3 = r2 + 2;
   ...
   r2 = r1 + 1;
   signal()
   ...
   store *q;
   signal_mem()
} while (condition)
```

(b) Loop parallelized with synchronization.



(c) Execution serialized due to synchronization.



(d) Parallel execution in TLS mode.

Figure 3.1: Using synchronization and speculation to satisfy inter-iteration data dependences.

The weight of each loop in an application is summarized as the *combined execution time coverage*, which is defined as the fraction of total execution time of the program spent on a particular loop. In this chapter, this weight is estimated using hardware performance counters. To accurately estimate the *combined* coverage of a set of loops, the nesting relationship of these

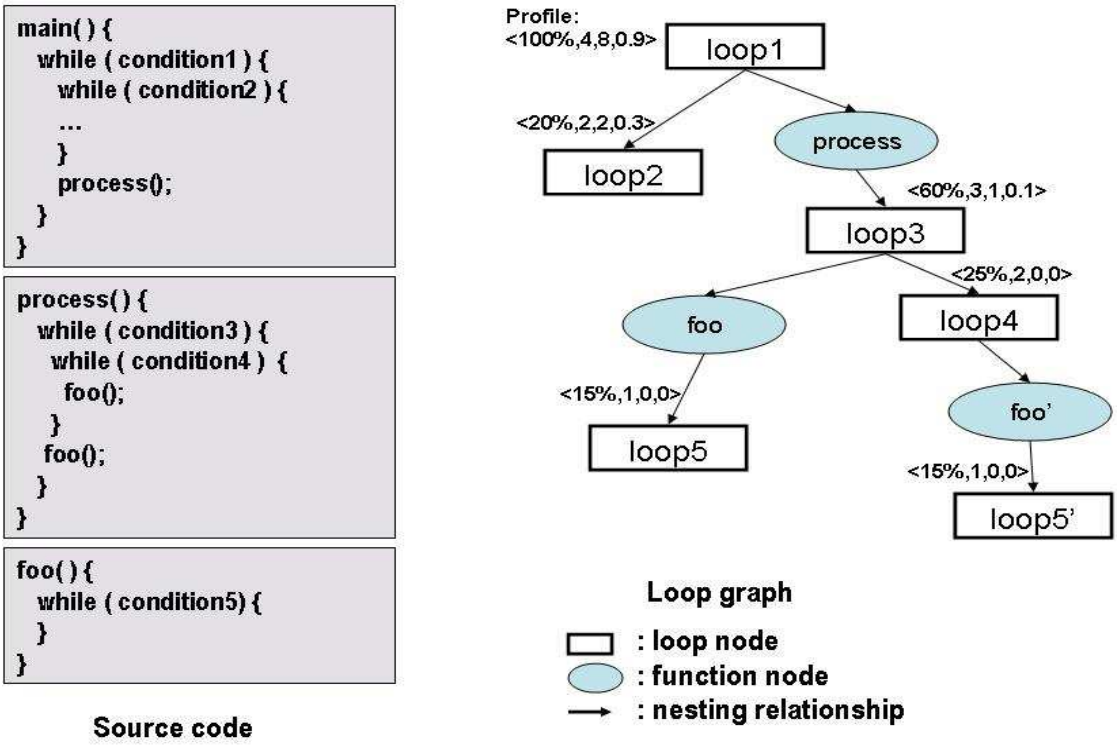


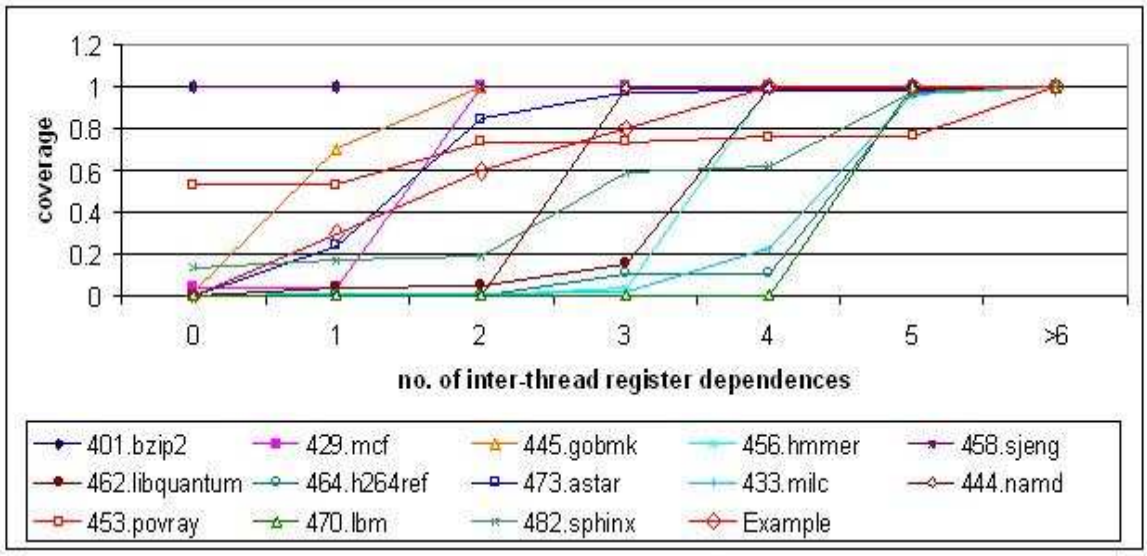
Figure 3.2: An example loop tree showing nesting relationship between loops. Each loop is annotated with four numbers: coverage, number of inter-thread register-based dependences, number of inter-thread memory-based dependences, and the probability of the memory-based dependence with highest probability.

loops must be determined—this is done with the help of a loop tree (for example, Figure 3.2). An example program and its corresponding loop structure along with profile information is shown in Figure 3.2. In the example, `loop4`, `loop5` and `loop5'` have no inter-thread memory-based data dependence. The *combined coverage* of loops with no memory-based data dependence is the cumulative coverage of `loop4` and `loop5`, which is 40%. (Coverage of `loop5'` is not included since it is nested inside `loop4`). The loop tree structure used in this chapter is similar to the loop graph described by Wang *et. al* [48], except for loops that can be invoked through different calling paths are replicated in loop tree. For example, `loop5` in Figure 3.2 is replicated, since two different call paths can both lead to the invocation of `loop5`.

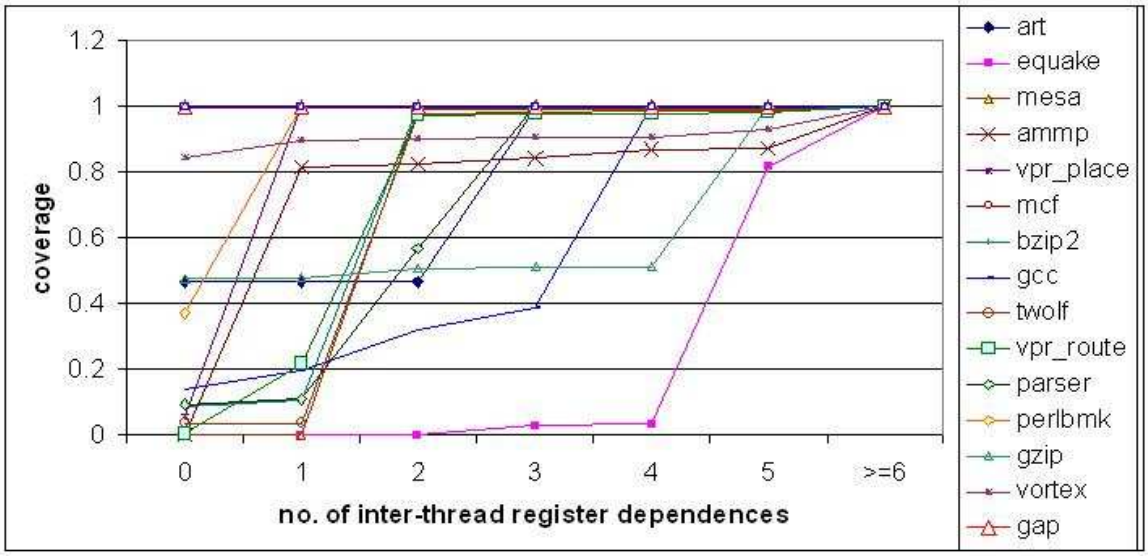
In this chapter, we consider the SPEC CPU 2006 and SPEC CPU 2000 benchmarks written in C or C++ (shown in Chapter 2 Table 2.1). We ignore the programs written in FORTRAN since they tend to be parallel scientific programs that can be successfully parallelized using traditional parallelizing compilers and do not require TLS support.

3.2.1 Inter-thread register-based data dependences

We first focus on the relatively straightforward register-based value dependences. For these dependences, the compiler is responsible for identifying instructions that produce and consume these value and generate synchronization to ensure correct execution. For example, in the loop shown in Figure 3.1(a), the compiler identifies the cross-iteration register-based dependence due to register $r2$ and inserts explicit synchronization, as shown in Figure 3.1(b). We count the number of inter-thread register-based dependences (true dependences) for each loop; and



(a) The combined execution time coverage for SPEC 2006 benchmarks.



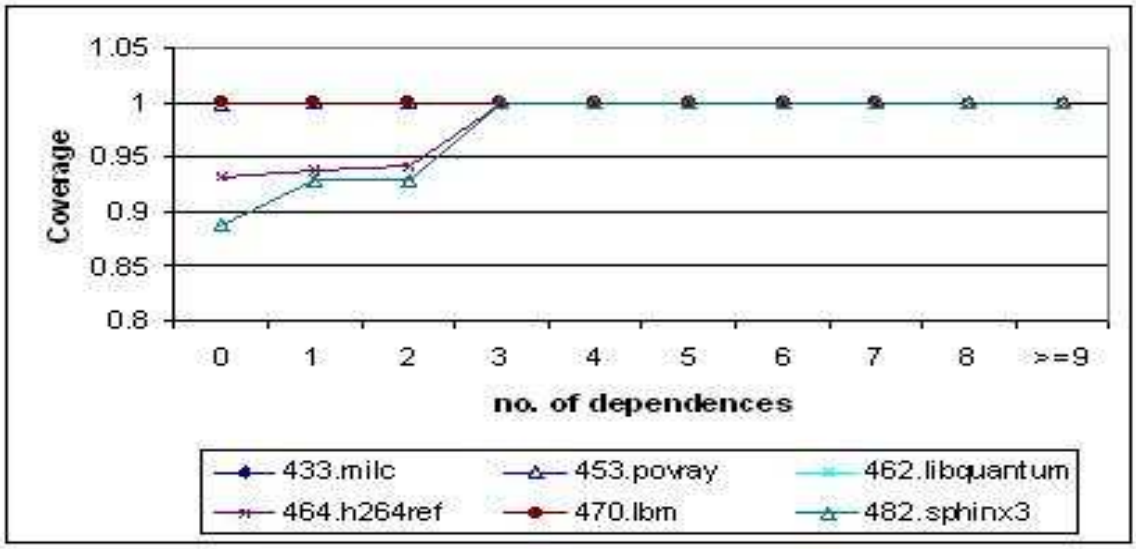
(b) The combined execution time coverage for SPEC 2000 benchmarks.

Figure 3.3: The combined execution time coverage of loops with inter-thread register-based dependences. The *x*-axis represents the number of register dependences and the *y*-axis represents the corresponding *combined* coverage estimated for the set of loops with *x* or lesser number of register dependences.

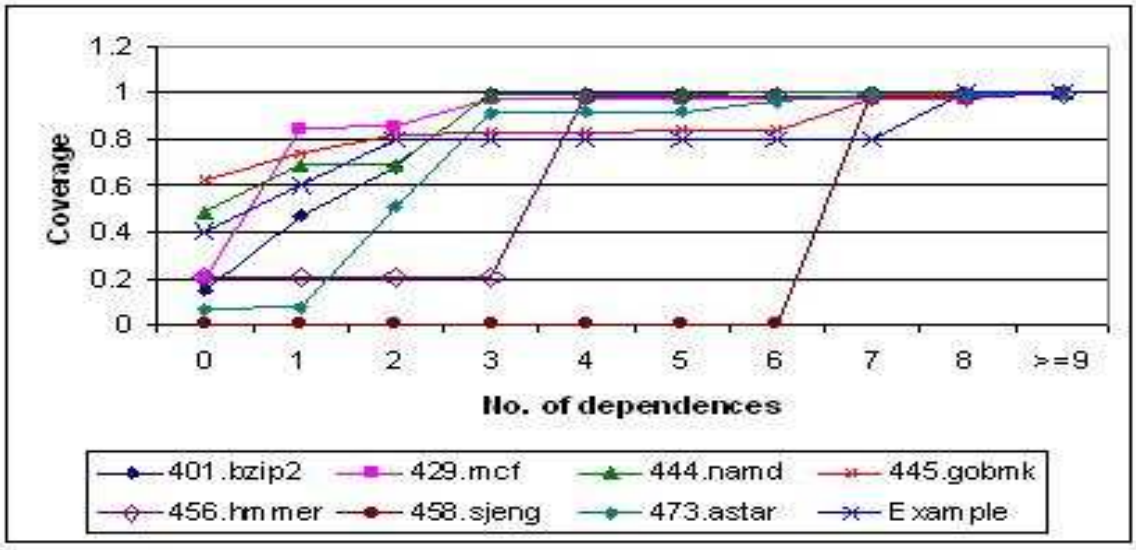
estimate the *combined* coverage of the set of loops with certain number of register-based dependences. The results for SPEC 2006 are presented in Figure 3.3(a) and the results for SPEC 2000 are shown in Figure 3.3(b). The *x*-axis represents the number of register dependences and the *y*-axis represents the corresponding *combined* coverage estimated for a certain set of loops. If a benchmark has a combined coverage of *C* for *x* number of dependences, it indicates that loops with less than *x* dependences have a combined coverage of *C*%. For example, for the loop-nest in Figure 3.2, the combined coverage of loops with 2 or fewer register dependences is 60%(coverage of loop2+loop4+loop5). The benchmarks with high combined coverage (*C*) for a small number of dependences (*x*), potentially exhibit high degree of parallelism. We found that the high coverage loops in most benchmarks have inter-thread register-based dependences. Thus, an effective TLS compiler that is capable of synchronizing a few inter-thread register dependences is essential. Zhai *et. al* [20] have described how such compiler optimizations can be implemented; and further shown that aggressive compiler scheduling techniques can reduce the critical forwarding path introduced by such synchronizations.

3.2.2 Inter-thread memory-based data dependences

Unlike register-based dependences, memory-based dependences are difficult to identify using a compiler due to potential aliasing. To ensure correctness, traditional parallelizing compilers insert synchronizations on all possible dependences. With TM or TLS support, the compiler is able to aggressively parallelize loops by speculating on ambiguous data dependences. However, the performance of such execution depends on the likelihood of such data dependences



(a) The combined execution time coverage for benchmarks with few inter-thread memory dependences. (Class 'A').



(b) The combined execution time coverage for benchmarks with inter-thread dependences. (Class 'B')

Figure 3.4: The combined execution time coverage of loops as a function of the number of inter-thread memory-based data dependences.

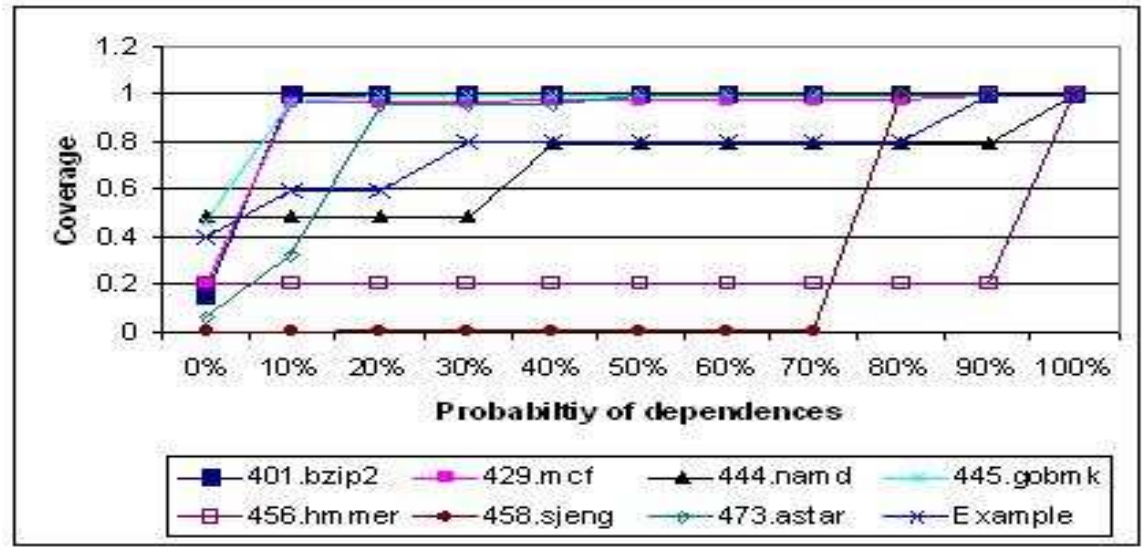


Figure 3.5: The coverage of loops with inter-thread memory-based data dependences less than a certain probability for Class 'B' benchmarks in SPEC 2006.

occurring at runtime. If a data dependence does occur, a thread can potentially violate data dependence constraints, and thus must be squashed and re-executed; recovery codes can be executed to restore correct state. For example, there is an ambiguous cross-iteration dependence, shown in Figure 3.1(a), due to load through pointer $*p$ and store through pointer $*q$. Although the compiler cannot determine whether there is a dependence between $*p$ and $*q$, it can obtain probabilistic information through data dependence profile. In this section, we conduct detailed analysis on inter-thread memory-based dependence using profiling information.

We classify benchmarks based on the combined coverage of loops with different number of memory-based dependences. First we present the results for SPEC 2006. Figure 3.4(a) shows the results of benchmarks (points corresponding to 433.MILC, 453.POVRAY, 462.LIBQUANTUM and 470.LBM in Figure 3.4(a) overlap) that can achieve a high combined coverage with

only a few inter-thread memory-based data dependences (Class ‘A’); Figure 3.4(b) shows the rest of the benchmarks (Class ‘B’). For benchmarks in Class ‘A’, 90% or more of the total execution can potentially be parallelized by only considering loops with no inter-thread dependences. These benchmarks can be parallelized without hardware support for speculative execution, if the compiler is able to prove independence between threads.

In Class ‘B’ benchmarks, the speculative hardware support are potentially useful, since inter-thread data dependences do occur. Figure 3.5 shows the probability of such data dependences and their corresponding coverage for Class ‘B’ benchmarks. The *x*-axis represents the probability of inter-thread memory-based dependences and the *y*-axis represents the corresponding *combined* coverage estimated for a certain set of loops. If a benchmark has a combined coverage of *C* for *x* probability of inter-thread dependence, it indicates the loops that only have inter-thread dependences with probability of less than *x* have a combined coverage of *C*%., For example, for the loop-nest in Figure 3.2, the combined coverage of loops with only 10% or lesser probability memory dependences is 80%(coverage of loop2+loop3). Other loops are nested inside loop3. Benchmarks 401.BZIP2, 429.MCF, 445.GOBMK and 473.ASTAR can achieve a large combined coverage, if all loops that only contain data dependences that occur in less than 20% of iterations are speculatively parallelized. These are the loops that could potentially benefit from TLS support.

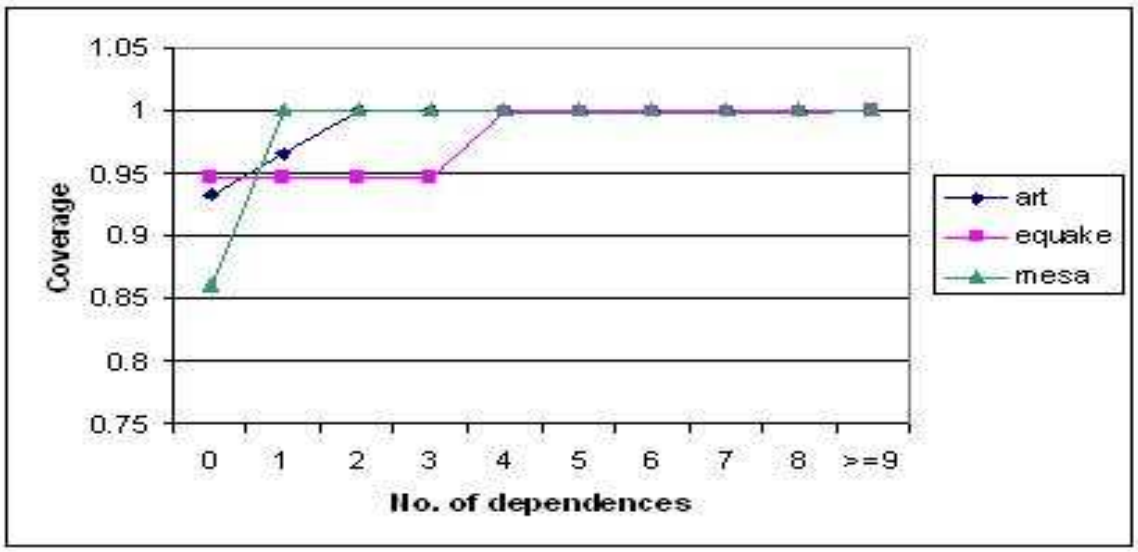
Some benchmarks, such as 456.HUMMER, 458.SJENG and 444.NAMD, can only achieve a high combined coverage, if loops containing frequently-occurring memory-based dependences are parallelized. These dependences potentially require synchronization. Previous studies has

shown that frequently occurring memory-based data dependences could be synchronized by the compiler with profiling data [21]; and aggressive code scheduling could reduce critical-path length introduced by such synchronizations [63].

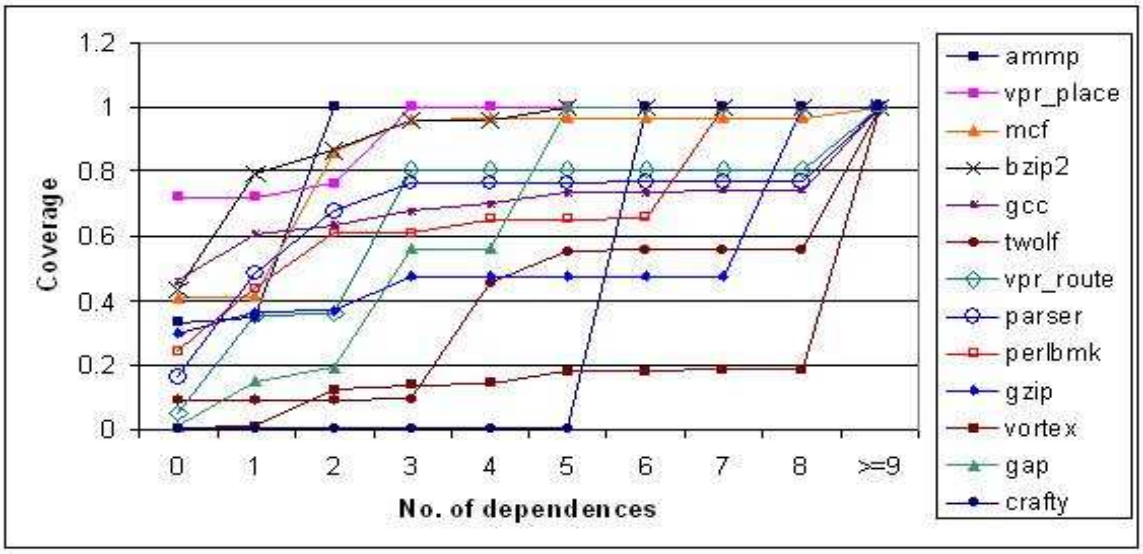
Figure 3.6(a) shows Class 'A' benchmarks in SPEC 2000—benchmarks with few inter-thread data dependences; Figure 3.6(b) shows Class 'B' benchmarks—benchmarks with several cross-iteration dependences. Comparing against SPEC 2006 results, shown in Figure 3.4(a) and in Figure 3.4(b), we found that SPEC 2000 suite has fewer Class 'A' benchmarks. Also the Class 'B' benchmarks in SPEC 2000 can only achieve high combined coverage by parallelizing loops with several cross-iteration dependences. Furthermore, by examining Figure 3.7, which presents the frequency of data dependences that must be speculated during parallel execution, we found that with the exception of AMMP, MCF, VPR_PLACE AND BZIP2, Class 'B' benchmarks in SPEC 2000 must speculate on high-probability cross-iteration dependences to achieve a high combined coverage. This is consistent with results reported by previous studies: in SPEC 2000, only a few benchmarks, AMMP, MCF, VPR_PLACE, demonstrated high degree of parallelism under TLS. The data dependence characteristics in SPEC2000 and SPEC2006 illustrate that SPEC 2006 can potentially achieve a higher degree of parallelism under the context of TLS.

3.2.3 Pitfalls

Even though profiling inter-thread data dependences is crucial in determining the suitability of using TLS to parallelize a loop, TLS performance cannot be directly inferred from this information. TLS performance depends on many other factors such as the size of the threads, thread



(a) The coverage for benchmarks with fewer inter-thread memory dependences. (Class 'A')



(b) The coverage for benchmarks with signifi cant inter-thread dependences. (Class 'B')

Figure 3.6: The coverage of loops with certain number of inter-thread memory-based data dependences in SPEC 2000

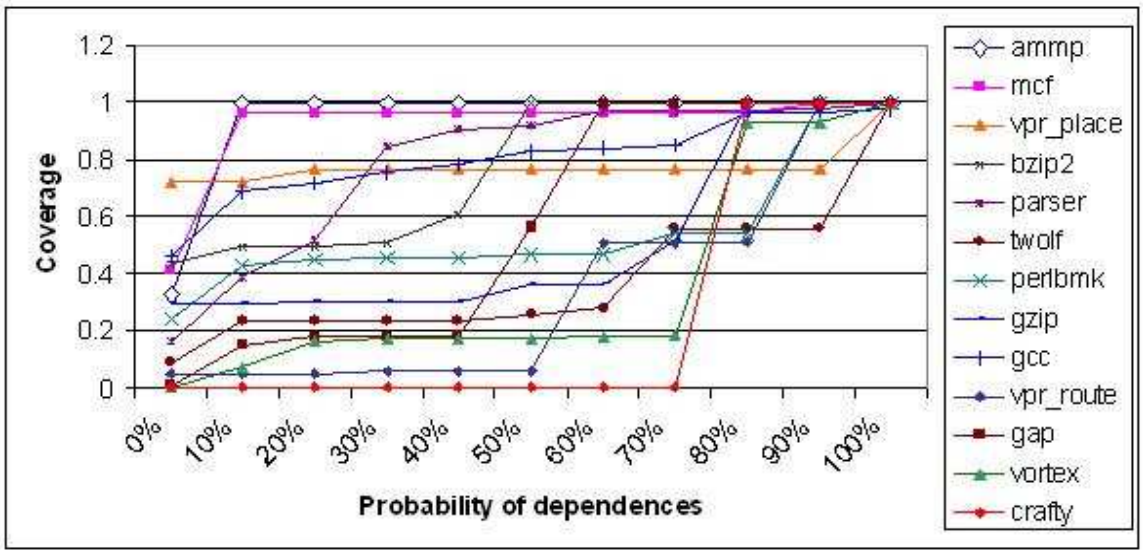


Figure 3.7: The coverage of loops with inter-thread memory-based data dependences less than a certain probability for Class-'B' benchmarks in SPEC 2000.

spawning overhead, loop iteration counts, etc; aggressive code scheduling can reduce the impact of synchronization for inter-thread dependences [20, 21, 48]. Furthermore, library calls can also cause inter-thread data dependences, which is not taken into account here. A common example is the call to *malloc*, which could potentially cause inter-thread dependences due to its internal data structures. Such dependences can potentially be eliminated using parallel libraries.

3.3 Compilation and evaluation infrastructure

To evaluate the amount of parallelism that can be exploited with hardware support for coarse-grain speculation and advanced compiler optimization technology in the SPEC 2006 and SPEC 2000 benchmark suites, we simulate the execution of these benchmarks with our detailed architectural simulator discussed in chapter 2.

Table 3.1: Architectural parameters.

Parameter	
Fetch/Issue/Retire width	6/4/4
Integer units	6 units / 1 cycle latency
Floating point units	4 units / 12 cycle latency
Memory ports	2Read, 1Write ports
Register Update Unit (ROB,issue queue)	128 entries
LSQ size	64 entries
L1I Cache	64K, 4 way 32B
L1D Cache	64K, 4 way 32B
Cache Latency	L1 1 cycle, L2 18 cycles
Memory latency	150 cycles for 1st chunk, 18 cycles subsequent chunks
Unifi ed L2	2MB, 8 way associative, 64B blocksize
Physical registers/thread	128 Integer and 128 Floating point registers
Thread overhead	5 cycles for fork/commit and 1 cycle for inter-thread communication
No. of cores	4

In this study, we use the reference input to simulate all benchmarks. In case of benchmarks with multiple input sets, the first input set is used. To get an accurate estimate of TLS performance, we parallelize and simulate all loops with at least 0.05% dynamic execution time coverage in all benchmarks. Based on the simulated speedup of each loop, we use our loop selection algorithm to select the best set of loops which maximizes the performance of the entire benchmark [48]. To report the speedup achieved by the entire benchmark, the average speedup of all the selected loops is calculated and weighted by the coverage of the loops. For each simulation run, several billion instructions are fast-forwarded to reach the loops and different samples of 500 million instructions are simulated to cover all the loops.

Table 3.2: Coverage of loops parallelized.

Benchmark	Coverage (%)			No. of loops		
	I	I + II	I + II + III	1	I + II	I + II + III
milc	13	79	79	5	22	22
lbm	0	100	100	0	1	2
h264ref	0	53	83	2	32	36
libquantum	0	98	98	1	5	5
sphinx3	40	83	91	11	19	21
povray	0	3	63	0	4	5
bzip2	2	3	31	4	6	14
mcf	0	85	93	0	6	6
namd	1	8	96	7	22	50
gobmk	0	6	13	0	1	5
hmmer	0	0	79	2	1	6
sjeng	0	0	1	0	0	6
astar	0	5	99	0	2	8

3.4 Exploiting parallelism in SPEC2006

In this section, we evaluate the amount of parallelism available in SPEC 2006 benchmarks using the framework described in Section 3.3. To isolate the parallelism that cannot be exploited without the help of TLS, we take three increasingly aggressive attempts to parallelize loops in SPEC 2006 benchmarks:

Type I: Loops that are identified as parallel by a traditional compiler;

Type II: Loops that have no inter-thread data dependence at runtime (for the particular *ref* input set used), but are not identified as parallel by the compiler, a.k.a., *Probably Parallel Loops*;

Type III: Loops that contain inter-thread data dependences, thus require TLS support to parallelize, a.k.a., *True Speculative Loops*.

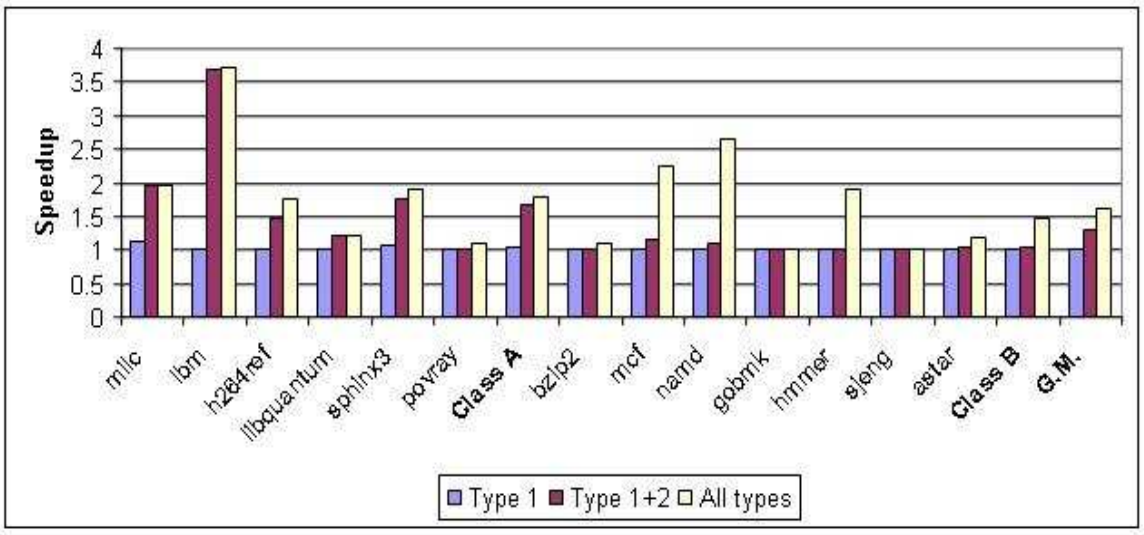


Figure 3.8: Shows the program speedup when different types of loops are parallelized using 4 cores.

Table 3.2 shows the percentage of total execution that can be parallelized when loops of different types become parallelizable.

To determine the performance impact associated with parallelizing a particular type of loops, the set of loops belong to that type are selected to maximize overall performance. The overall program speedup is then calculated by considering the speedup and coverage of the selected loops. For example, let the selected set of loops be $\{L_1, L_2, L_3, ... L_n\}$. Let their corresponding coverage be $\{ C_1, C_2, C_3, ... C_n \}$ and their corresponding speedup be $\{ S_1, S_2, S_3, ... S_n \}$. The overall program speedup is then calculated as $Speedup = 1/((1-(C_1 + C_2 + .. C_n)) + C_1/S_1 + C_1/S_2 + .. C_1/S_n)$. In this experiment, we assume it is always possible to identify the optimal set of loops that maximize overall performance, however, in reality, the compiler can potentially select sub-optimal loops due to inaccurate performance estimation [63].

3.4.1 Type I loops

We applied the Intel C++ compiler [64] to the SPEC 2006 benchmarks to select parallel loops. The benchmarks are compiled with the following options: `-O3 -ipo -parallel -par-threshold0`. The option `-par-threshold0` allows the compiler to parallelize loops without taking into consideration thread overhead. The loops selected by the Intel compiler are then parallelized using our TLS compiler and simulated. The speedup achieved by the selected loops over sequential execution is shown as the first set of bars in Figure 3.8. With the exception of MILC, which achieved a speedup of 11%, and SPHINX3, which achieved a speedup of 7%, none of the benchmarks is able to speedup over the sequential execution. Overall, the geometric mean of the speedup is only 1%.

This result is anticipated, since the complex control flow and ambiguous data dependence patterns prohibit the traditional compiler from parallelizing large loops. We have found that in most benchmarks the compiler only choses to parallelize simple inner loops with known iteration count. It is worth pointing out that, although many class **A** benchmarks, such as MILC and LBM, contain loops with no inter-thread data dependences, the compiler is unable to identify these loops as being parallel.

3.4.2 Type I + II loops

With the addition of *Probably Parallel Loops*, class **A** benchmarks achieve significant performance gain, however, class **B** benchmarks remain sequential. The class **A** benchmarks gain 68% speedup due to these *Probably Parallel Loops* while class **B** benchmarks gain only 4%.

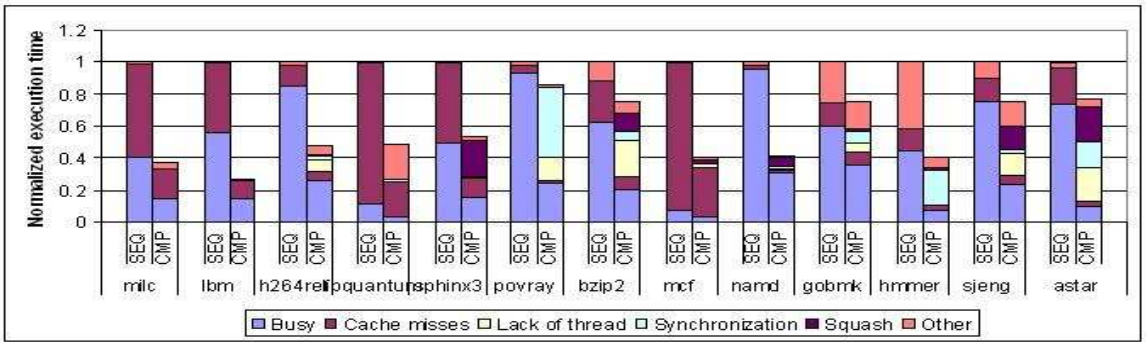


Figure 3.9: Shows the breakdown of execution time for loops normalized to sequential execution time

If the compiler is able to determine that these loops are parallel, we can potentially parallelize these loops without TLS support. Among the class **A** benchmarks, significant portion of the loops in SPHINX3 and H264REF are *Probably Parallel Loops*; and all loops in MILC, LBM and LIBQUANTUM are *Probably Parallel Loops*.

3.4.3 Type I + II + III loops

With the addition of *True Speculative Loops*, we find that many class **B** benchmarks are able to achieve speedup. With only these *True Speculative Loops* class **B** benchmarks gain a speedup of 42% giving them an overall speedup of 46%.

To examine TLS performance in detail, Figure 3.9 shows the execution time breakdown of parallel execution with TLS support (only selected loops) and sequential execution. The **SEQ** bars show the normalized execution time of the sequential execution running on one core. The **CMP** bars show the normalized execution time of the parallel program executing on four cores. Each bar is divided into six segments: *Busy* represents the amount of time spent in executing

useful instructions and the delay due to lack of instruction level parallelism inside each thread; *Lack of threads* represents the amount of time wasted due to the lack of parallel threads (probably due to low iteration count in a loop); *Synchronization* represents the amount of time spent in synchronizing frequently occurring memory dependences and register dependences; *Cache misses* represents the amount of time the processor stalled due to cache misses; *Squash* represents the amount of time wasted executing instruction that are eventually thrown away due to failed speculation; *Other* corresponds to everything else. In particular, it includes time wasted due to speculative buffer overflow and load imbalance between consecutive threads.

We will first focus on the class **B** benchmarks. In HMMER, the loop at `fast-algorithms.c:133` is selected for parallelization, however it has many inter-thread dependences that require synchronizations. These synchronizations create a critical forwarding path between the threads and serialize execution. Thus, by performing speculative instruction scheduling to move the producers of these dependences as early as possible in the execution [20, 48], the parallel overlap is significantly increased; and the benchmark achieves a 90% program speedup. Similar behavior is observed in NAMD, where synchronization and instruction scheduling leads to a 164% program speedup.

For ASTAR, the important loop is at `way2_.cpp:100`, which has a few inter-thread dependences. Some of these dependences are frequent, and thus are synchronized; others are infrequent, and thus are speculated on. Without TLS support, these infrequent occurring dependences must be synchronized, and can lead to serialization of the execution. With the help of TLS, this loop achieves a 17% speedup.

POVRAY, although a class **A** benchmark, is able to benefit from speculation. The important loop in `csg.cpp:248` is a *true speculative loop* with a few mis-speculations, thus it is non-parallel for a traditional compiler. Unfortunately, the selected loops have small trip counts, and the cores are often idle; thus the benchmark is only able to achieve a moderate program speedup of 9%.

Not all benchmarks are able to benefit from TLS. GOBMK has many loops with low trip counts, thus many execution cycles are wasted as the cores are idling. Even the loops with large trip counts are not able to achieve the desired speedup for two reasons: first of all, the amount of work in consecutive iterations is often unbalanced; secondly, many iterations have large memory footprints that lead to buffer overflow of the speculative states. The geometric mean of the thread size for the top 50 loops (in terms of coverage) is 800,000 instructions. Overall, GOBMK only achieves 1% performance improvement with TLS support.

Loops in SJENG have many inter-thread dependences that occur in 70% of all iterations, and thus need synchronization. However, the critical forwarding path introduced by these synchronization cannot be reduced through instruction scheduling due to intra-thread dependences. Thus, SJENG was unable to benefit from TLS.

To summarize, TLS is effective in parallelizing both class **A** and class **B** loops. Overall, if we select the optimal set of loops, we can achieve a program speedup of about 60% (geometrical mean) , in contrast to a traditional compiler, which only achieves a 1% program speedup.

Table 3.3: Coverage of loops parallelized in SPEC 2000.

Benchmark	Coverage (%)			No. of loops		
	I	I + II	I + II + III	1	I + II	I + II + III
mesa	0	4	37	0	6	7
art	7	79	79	2	7	7
equake	14	90	90	4	9	10
gzip	0	9	59	0	1	5
vpr_place	1	2	80	0	1	7
vpr_route	0	25	74	0	3	7
gcc	7	9	27	2	11	25
mcf	0	0	91	0	2	6
crafty	0	0	1	0	0	3
ammp	0	15	48	0	3	7
parser	0	23	51	0	21	33
perlbmk	0	7	57	0	2	9
gap	0	0	30	0	0	2
vortex	0	8	20	0	2	2
bzip2	0	0	81	0	0	19
twolf	0	17	26	0	5	5

3.5 Comparison with SPEC2000

In this section, we evaluate the amount of parallelism available in SPEC 2000 and contrast it with results from SPEC 2006. Table 3.3 shows the percentage of total execution that can be parallelized when loops of different types become parallelizable for SPEC 2000 benchmarks. Figure 3.10 shows the speedup for SPEC 2000 benchmarks due to different types of loops.

As in the case of SPEC 2006 the compiler fails to identify parallel loops, except for two Class-'A' benchmarks ART and EQUAKE, leading to only 2% overall performance due to Type-I loops.

Figure 3.10 shows that *Probably Parallel Loops* achieve a geometric mean speedup of 10%. Among the Class-'A' benchmarks, almost all the loops do not suffer from any mis-speculations and thus are *Probably Parallel Loops*. But among Class-'B' benchmarks which have inter-thread dependences, the geometric mean speedup due to *Probably Parallel Loops* is only 3%.

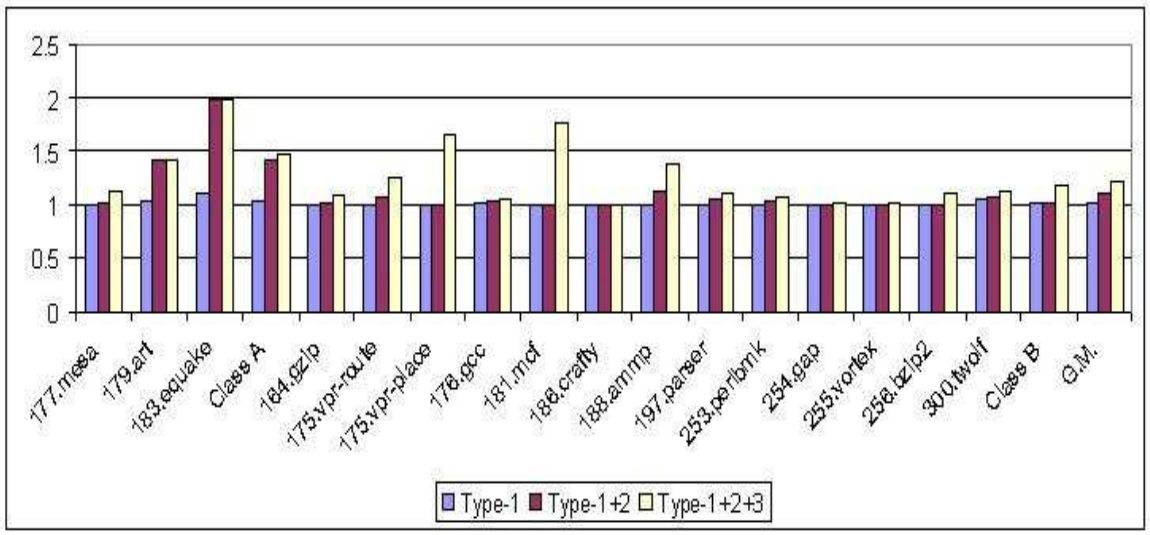


Figure 3.10: Shows the program speedup when different types of loops are parallelized using 4 cores for SPEC 2000.

With the addition of *True Speculative Loops*, the speedup for Class-'B' increases to 19%. This shows the importance of supporting TLS to exploit all the potential parallelism in SPEC 2000 benchmarks.

When compared to SPEC 2006 benchmarks the overall speedup is only 24% where the SPEC 2006 achieved 60%. This clearly shows the trend toward more parallelism in more recent class of applications in SPEC 2006 suite.

3.6 Conclusions

Previous studies of SPEC 2006 based on high level analysis have shown only a limited potential for TLS. These studies did not taken into account the benefits of compiler-based optimizations. In this chapter, using a state-of-the-art TLS compiler, we show that SPEC 2006 applications can

be successfully parallelized with TLS.

We show that often the traditional parallelizing compiler cannot prove independence due to the existence of complex control flow and ambiguous data accesses, even if many benchmarks contain parallel loops. With the help of TLS, these *potentially parallel loops* can be parallelized, and thus potentially allowing six benchmarks, MILC, LBM, H264REF, LIBQUANTUM, SPHINX and POVRAY, to achieve a speedup of 78%, if the best set of loops are selected. Furthermore, TLS can parallelize loops that cannot be parallelized by traditional compilers due to infrequent inter-thread dependences (*true speculative loops*). With TLS, benchmarks BZIP2, MCF, NAMD, GOBMK, HMMER, SJENG and ASTAR can potentially achieve an additional 46% speedup. Overall, with four cores we can achieve a speedup of 60% on all benchmarks (geometric mean) and with eight cores the speedup can reach 91% when compared to sequential execution.

When compared to SPEC 2006, the SPEC 2000 benchmarks have more inter-thread dependences leading to fewer class 'A' benchmarks and overall performance of only 26% when compared to 60% in SPEC 2006. This shows a trend toward more parallel applications and the need to support TLS in future multi-threaded/multi-core architectures to exploit this available parallelism.

Chapter 4

TLS support in SMT

Most previous work on TLS assumed CMP based architectures while only a few have concentrated on SMT based multi-threaded architectures. Existing SMT based speculative multithreading approaches either use complex hardware [65] or use limited resources like Load-Store Queues(LSQs) [7, 30] to buffer speculative results, and to record load addresses to check for dependence violations. The advantage of LSQ-based method is that the LSQs are already available in most modern processors, so no major modifications to the processor architecture is needed. The main disadvantage of a LSQ-based implementation is the limited buffer. Due to this consideration, LSQ-based architectures can support only small threads. But previous research [48, 20] have shown that when we need to consider a more realistic overhead of forking a thread, it becomes important to support larger threads in order to amortize the TLS overhead. Also loops suitable for TLS parallelization could be present at outer loops that have larger iteration size. Hence, it is important to support larger threads.

In this chapter, we propose a novel cache-based architecture to implement speculative multithreading in SMT processors that only requires adding a few extra bits to each cache line in existing L1 cache. The proposed approach is able to handle large threads since the entire L1 cache can be used to buffer speculative states.

4.1 Related work

Hardware support for speculative multithreading architectures have been studied intensely during the past decade. Earlier architectures were based on special hardware structures for dependence checking like the address resolution buffer (ARB) in [41], and the memory disambiguation table (MDT) in [66]. These special hardware structures are of limited size and need extra cycles to access them. To avoid these limitations cache-based architectures like speculative versioning cache (SVC) [42] and STAMPede [17] were proposed.

When compared to speculative multithreading on chip multiprocessors (CMPs), there are relatively few studies on supporting speculative multithreading for SMTs. In [67], private L1 cache for each context is used to buffer speculative values and do dependence checking. In DMT [7] and in IMT [30], enhanced LSQs are used.

The main limitation of the LSQ-based approach is the limited size of the queue. To overcome this limitation we propose a cache-based scheme in this chapter. We draw many ideas from the cache architectures proposed for CMPs. The difference is that the CMP-based architectures have private L1 cache for each core and is used to buffer results. The dependence

checking hardware is also distributed among different L1 caches. In our approach, all the contexts in the SMT share the same cache.

Concurrent to our work, STAMPede [68] has extended the cache protocol described in [17], to support shared cache architectures. Their technique was studied in the context of multicore processors using shared cache. In [69], shared L2 cache based technique was used to speculatively parallelize database applications. Though they mention that it could be applied to SMT processors all their results and conclusions are for CMPs, while our scheme is specifically aimed at SMT processors.

4.2 SMT model

We consider a SMT architecture where many resources like fetch queue and issue queue are fully shared [70]. Figure 4.1 gives a block diagram of the SMT architecture. When more then one thread are actively executing, we need to choose which thread to fetch instructions from in every cycle. We use the ICOUNT policy shown in [70] to decide on the thread to fetch from. Also when instructions from multiple threads are ready to commit, the instructions from non-speculative thread is given more priority.

To implement speculative multithreading, we need hardware support to buffer results from speculative threads and detect dependence violation between threads. In section 4.3, we first present a simplified scheme that supports only one speculative thread, and in section 4.4 we extend this scheme to four (or more) threads.

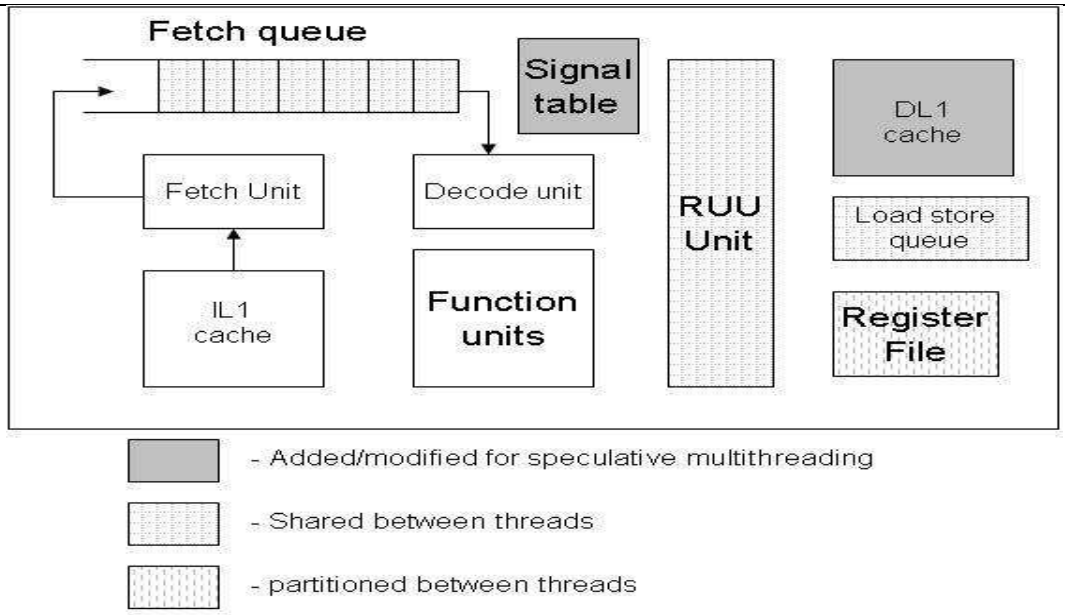


Figure 4.1: SMT Block Diagram

4.3 Simplified two-thread scheme

In this section, we consider a SMT processor with only two threads. As there are only two threads, at least one of the threads has to be non-speculative, i.e. there will be at most one speculative thread. In such a two-thread SMT, we only need to introduce two extra states to each cache line - *Speculative Valid (SV) and Speculative Dirty (SD)*. Each cache line also needs two extra bits - *Speculative Load (SL) and Speculative Modified (SM)* to support data dependence checking. In the proposed scheme, all of speculative data are kept only in the shared L1 cache, and all of the data stored in L2 cache are non-speculative. Figure 4.2 presents the cache-line state transitions in this scheme. In Figure 4.2 the transitions are of the form 'Command from processor / Action taken'. The processor can issue load, store, speculative load and speculative store commands to the L1 cache.

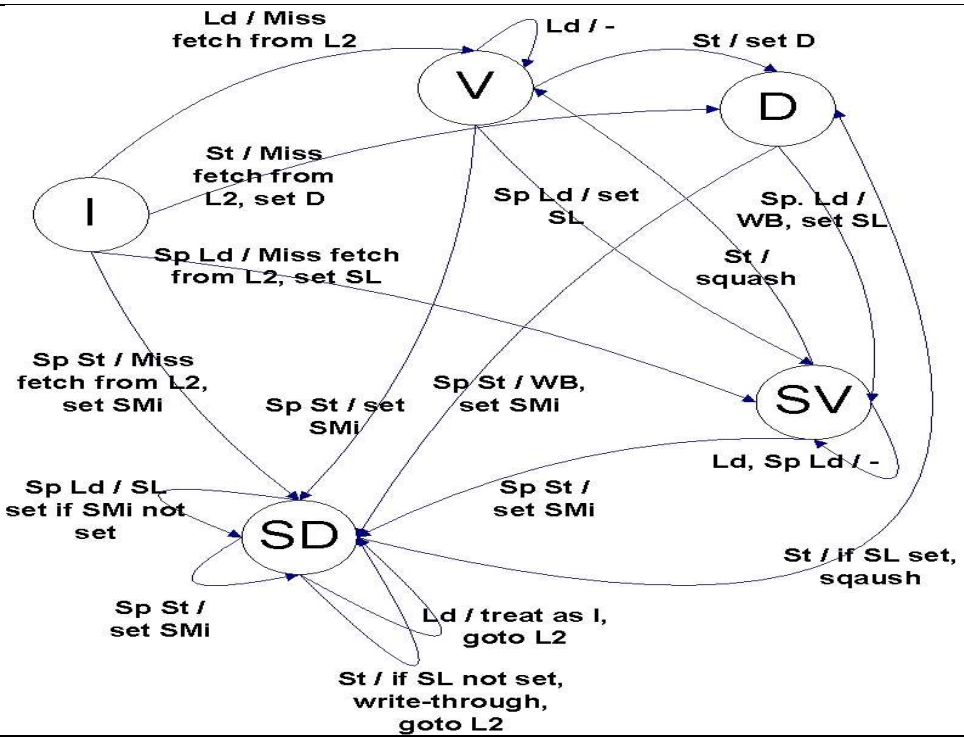


Figure 4.2: Two Thread Scheme - Cache State Transitions

Speculative value buffering When a speculative thread writes, the value is stored in the shared L1 data cache with the SM bit of the cache line set and the cache line transitions to the SD state. The value stays in the cache till the thread is committed or squashed. Thus, the L1 D-cache acts as a *store buffer* that stores speculative updates.

Dependence Violation Detection When a speculative thread issues a load operation, it first checks whether a speculative thread has already written the value. However, by having just one SM (speculative modified) bit for each cache line, we cannot be sure which word in a particular cache line was written by the speculative thread. To allow more precise dependence information, we could maintain one SM bit (SMi) for each word in the cache

line. If the SMi bit is not set, the SL (speculative load) bit will be set and the cache line transitions to SV (speculative valid) state, as this load could cause a possible dependence violation, when a non-speculative write arrives later.

Here, when a non-speculative thread writes into a cache line, if the SL bit is already set, it indicates that the speculative thread has read a stale value. The speculative thread will be squashed and restarted.

Non-speculative thread execution If the state of the cache line being written to is SD (speculatively dirty), the non-speculative thread writes the value directly to L2 cache. Also, it writes the portion of the data non-overlapped with the speculatively modified data (indicated by SMi bits) into the L1 cache. This merging is done, so that the speculative thread can get the most recent non-speculative value from L1 cache. Also this simplifies the commit operation.

Reads by a non-speculative thread to a speculatively modified line (SD) are treated as a *cache miss*. While handling this cache miss the non-speculative thread takes the value directly from the L2 cache.

Replacement policy Speculatively modified cache lines or the lines with the SL bit set cannot be evicted from the cache. If evicted, we lose information which can lead to incorrect execution. When we have to replace a line, a line which has none of the SL and SM bits set is selected.

If a non-speculative thread needs to replace a line and couldn’t find a clean line, it avoids

replacing the speculative line by directly sending the request to L2 cache. In case of speculative thread, the thread is suspended. Once the speculative thread becomes non-speculative, the SL and SM bits are cleared to allow it to continue its execution.

Commit and Squash When a thread commits, both the SL and SMi bits are cleared. This can be easily implemented as a gang-clear operation. Unlike other schemes where every speculative value needs to be written to the cache at the point of commit (which could potentially take hundreds of cycles), the commit operation can be done in just one cycle in our scheme by gang-clearing both SL and SMi bits.

When a thread squashes, the SL bit in all cache lines is cleared (gang-clear). The valid bit for a cache line is also cleared if the SM bit is set. This is like the conditional gang-clear operation used in Cherry[71]. It was shown that this operation can be easily implemented in only a few cycles.

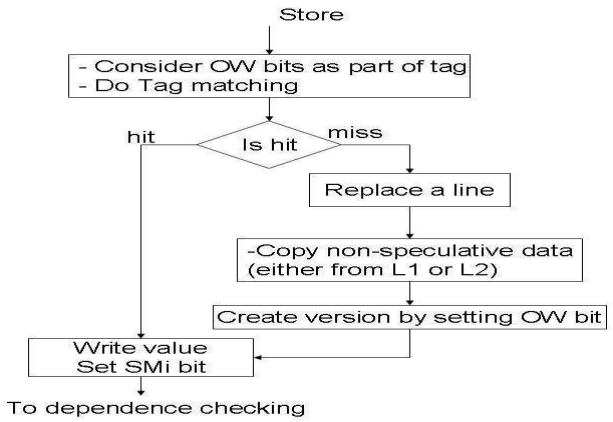
4.4 Four-thread scheme

When executing more than one speculative thread, the L1 D-cache needs to buffer results from two or more threads, so the two-thread scheme cannot be directly applied. In this section we propose a scheme which can efficiently handle more than one speculative thread. The basic idea is to use the entire set in the cache to buffer different versions of the same line created by the different threads. We will use a 4-thread system to simplify our explanation. The scheme could be similarly extended to systems with more than 4 threads.

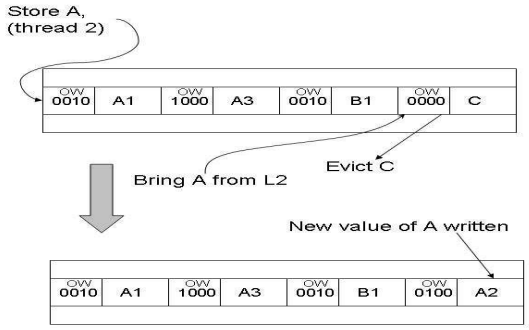
Speculative buffering The L1 D-cache has to buffer results from multiple threads, so we need to maintain different versions of the same cache line. All of the versions are kept on the same set in the cache. We introduce *Owner* bits (OW) which keep the speculative thread-id that wrote into the cache line. We need four OW bits for the four threads. For a non-speculative cache line, the OW bits are cleared. Buffering of speculative values is explained in Figure 4.3(a). Figure 4.3(b) shows an example where thread 2 tries to write a new version of A to a set which already contain versions from thread 1 and thread 3.

Speculative load execution A cache line can be read by any of the four threads, so a single SL bit is not sufficient to indicate which thread has caused dependence violations. We introduce a SL bit for each thread on each line of cache (4 bits for 4 threads). The execution of a *speculative load* instruction is explained in Fig 4.4. We can see that the *speculative load* can either load from its own version (i.e., a hit), from predecessor thread's version (i.e., a partial hit) or from the L2 cache (i.e., miss - Figure 4.5(b)).

Dependence detection When a store executes, it checks whether the versions of the cache line belong to any of its own successor threads. If SL bit is set for any of the successor threads, the successor thread is squashed along with its successors. The oldest squashed thread is then restarted. In case if the SL bit is not set, the store updates the latter thread's version if the corresponding SMi bit is not set. This is done so that the latter thread would get the latest version of the value stored.



(a) Method



(b) Example

Figure 4.3: Speculative Store Handling

Non-speculative thread execution Execution of a load in a non-speculative thread is very similar to the speculative load shown in Figure 4.4. But the non-speculative load does not set any SL bit and also the partial hit scenario does not occur. The execution of a non-speculative store is also similar to the speculative thread shown in Figure 4.3, except that the non-speculative store does not set the OW and SMi bits. Also, the non-speculative store merges its value with all versions in the cache. This is done so that the speculative

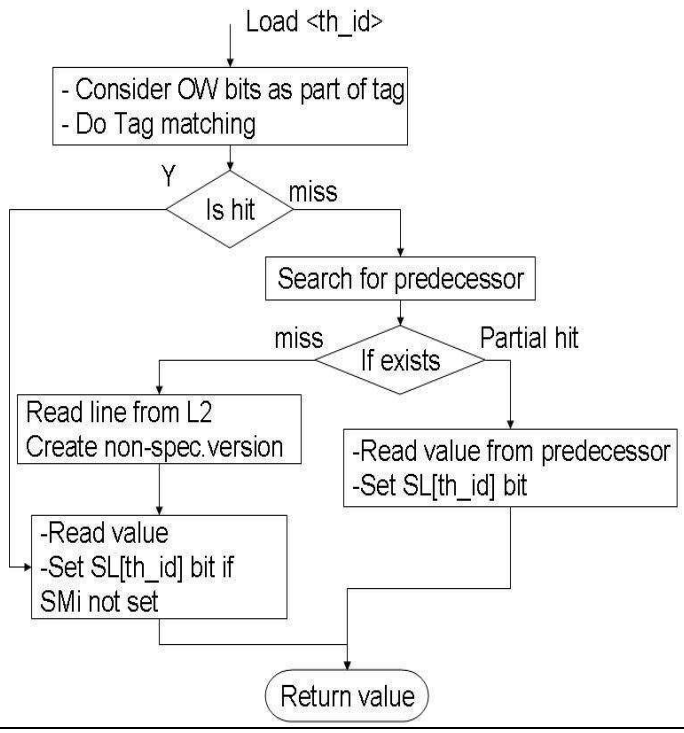


Figure 4.4: Speculative Load Handling

threads will get the most recent non-speculative version of the cache line.

Commit and squash To squash a thread, the SL[thread_id] is cleared for all of cache lines. This can be done as a gang-clear operation. A line is invalidated if any of its SMi bit is set. This is accomplished by a conditional gang-clear operation as in two-thread scheme. To commit a thread, the SL [thread_id] bit and the SMi bits of the thread are cleared. The commit operation must ensure that there is only one non-speculative version present in L1 cache. If a cache line to which the current thread wrote has another version which is earlier than that of the current thread, then that version needs to be written back and invalidated. To speedup the commit process, a list of blocks that needed to be committed

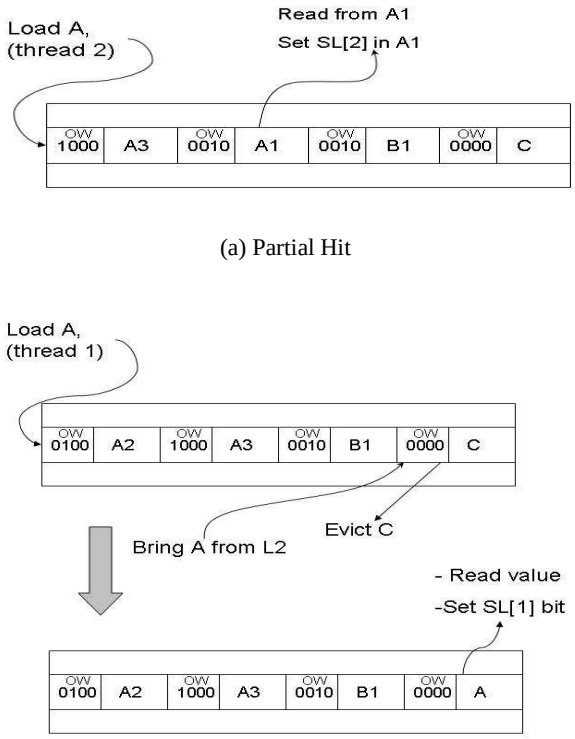


Figure 4.5: Speculative Load Handling Example

is maintained in a special hardware buffer called *owner* buffer. The *owner* buffer used is similar to the *owner* buffer assumed in [49] to assist the commit process. In addition to using *owner* buffer, we can potentially overlap the commit process with the execution of the next thread. Our simulation shows that this overhead causes no potential performance degradation.

Speculative State Overflow As we see in the two-thread case, we cannot replace a line with SL or SM bit set. If a speculative thread encounters a cache miss and if it is not able to

find a clean line to replace from the cache, it can either suspend and wait till it becomes non-speculative or it can squash the successor threads and consume its cache lines. In this thesis we avoid frequent squashes due to such *overflow* by forcing the speculative thread to stall till it becomes non-speculative. While waiting, a thread occupies shared resources like fetch queue, RUU and LSQ. There may be a situation where all the resources are occupied by the suspended thread and the non-speculation thread is unable to proceed, thus, causing a deadlock. To avoid this scenario, the speculative thread will give up its resources when it is stalled.

Speculative victim buffer In our approach, we use the different cache lines in the same set to buffer the speculative values. When two or more data locations are mapped to the same set, all the speculative versions (from 4 or more threads) cannot be buffered in the same set leading to overflow. As we saw above such speculative state overflow can lead to stalling of speculative threads. To reduce the impact of such overflow due to conflict cache misses, we introduce a *Speculative Victim Buffer*. Similar to a traditional victim buffer [72] the *Speculative Victim Buffer* buffers the cache blocks which are replaced from the cache. Unlike traditional victim buffers, the blocks evicted could contain speculative bits which need to be maintained till the corresponding speculative thread is committed. Every speculative load, in addition to the cache set in the L1 cache, also checks the *Speculative Victim Buffer* to get the most recent version. If the version is found in the victim buffer, the corresponding SL bit is set. Every store request searches the *Speculative Victim Buffer* to check for any dependence violations. Also during commit process, the

speculative versions of the committing thread found in the *Speculative Victim Buffer* are also committed.

To further reduce the stalling due to speculative state overflow, when a non-speculative thread misses in L1 data cache and all the versions available in the set are speculative, we force the non-speculative thread's request to directly go to the L2 cache.

Implementation issues While executing a *speculative load*, we may have to search the entire set in the cache to get the predecessor thread's cache line. Also, while detecting mis-speculation, we need to search the entire set to find if any successor thread has set the SL bit. These operations can be implemented by adding more logic to the tag matching hardware but it could increase cache hit time. In our scheme, we assume there is special hardware that does these "whole-set" operations, which is kept separate from the tag matching hardware. We need only one instance of this hardware and the whole cache set is copied into it when we have to perform such whole-set operations. We assume such special operations take 3 cycles.

4.5 Performance evaluation

In this section we compare the performance of LSQ based TLS architecture with the cache based two-thread and four-thread schemes proposed in the previous sections.

4.5.1 Experimental methodology

We used the simulation framework described in Chapter 2. The specific processor parameters used are described in Table 4.1.

Table 4.1: Architectural parameters.

Parameter	
Fetch/Issue/Retire width	6/4/4
Integer units	6 units / 1 cycle latency
Floating point units	4 units / 12 cycle latency
Memory ports	2Read, 1Write ports
Register Update Unit (ROB,issue queue)	128 entries
LSQ size	64 entries
Memory ports	2 read and 1 write ports
L1I Cache	64K, 4 way associative, 32B blocksize
L1D Cache	64K, 4 way associative, 32B blocksize
Cache Latency	L1 1 cycle, L2 18 cycles
Memory latency	150 cycles for 1st chunk, 18 cycles subsequent chunks
Branch predictor	Bimod, 2K entries
Unifi ed L2	4MB, 8 way associative, 64B blocksize
Branch mis-prediction penalty	6 cycles
Physical registers/thread	128 Integer and 128 Floating point registers
Thread overhead	5 cycles for fork/commit and 1 cycle for inter-thread communication
No. of cores	4

4.5.2 Results

We consider the following configurations:

SEQ: This is an out-of-order superscalar processor with parameters described in Table 5.1.

SMT-2: This is an out-of-order SMT processor which can support two threads at a time using the two-thread scheme described in Section 4.3. This configuration has the same number

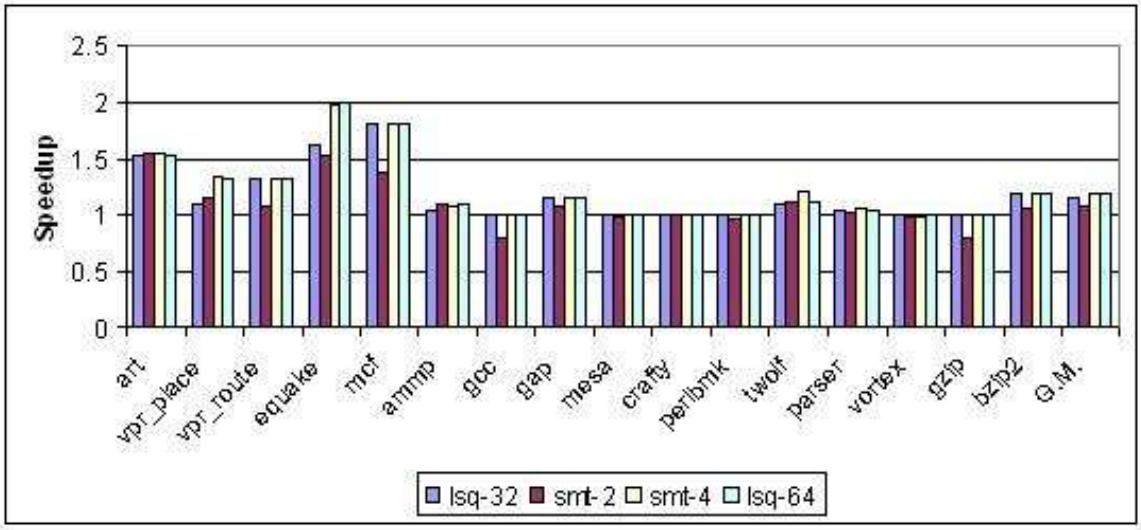
of functional units as in the superscalar.Each line of cache has 9 extra bits (8 SMi and 1 SL).

SMT-4: This SMT processor can support four threads using the four-thread scheme described in Section 4.4. It also has the same number of functional units as in SEQ. Each line of cache is augmented with 8 SMi bits, 4 OW bits and 32 SL bits (8 bits for each thread to avoid thread violations due to aliasing).

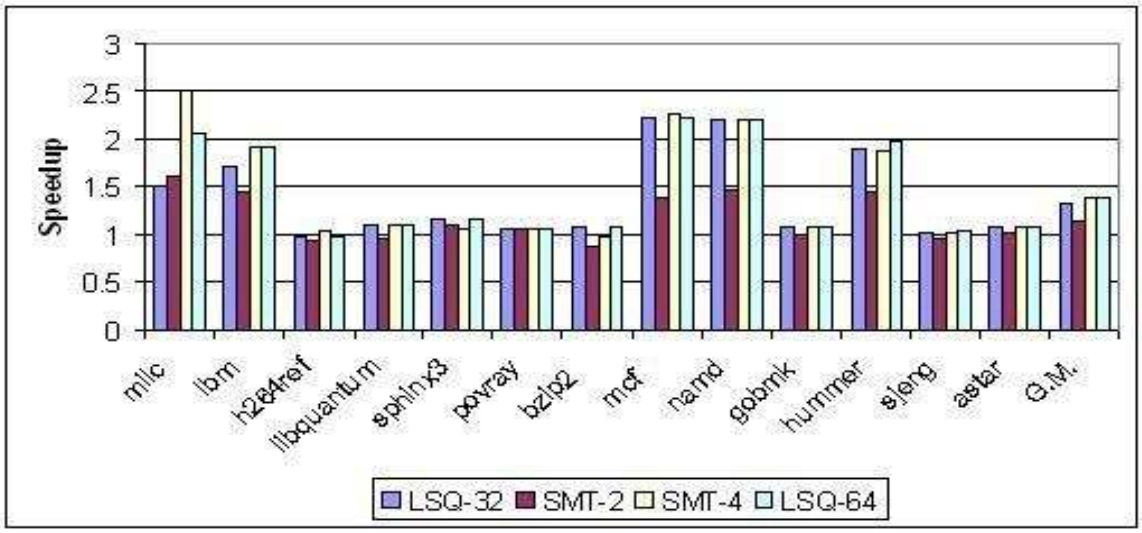
LSQ-32: This SMT processor supports 4 threads and uses the LSQ-based mechanism as in [30][7]. It has the same number of functional units, but uses extra space for enhanced LSQs that support speculation. Each thread has 32 LSQ entries. This is similar to the configuration used in previous studies [30, 7].

LSQ-64: This SMT configuration is similar to LSQ-32 except that it has 64 LSQ entries for each thread. We use this configuration to show the impact of using very large LSQs to implement TLS.

Figure 4.6(a) and Figure 4.6(b) shows the speedup of the four TLS architectures over the SEQ architecture for SPEC 2000 benchmarks and SPEC 2006 benchmarks respectively. We see that the SMT-2 architecture performs about 8% and 18% worse than the LSQ-32 architecture for SPEC 2000 and SPEC 2006 benchmarks correspondingly. Over all the benchmarks it performs about 12% worse than the LSQ-32 architecture. SMT-4 out performs LSQ-32 by 4% and 5% for SPEC 2000 and SPEC 2006 benchmarks respectively. Over all it performs 4% better than LSQ-32 architecture. The LSQ-64 architecture matches the performance of the SMT-4 architecture



(a) Speedup for SPEC 2000 benchmarks.



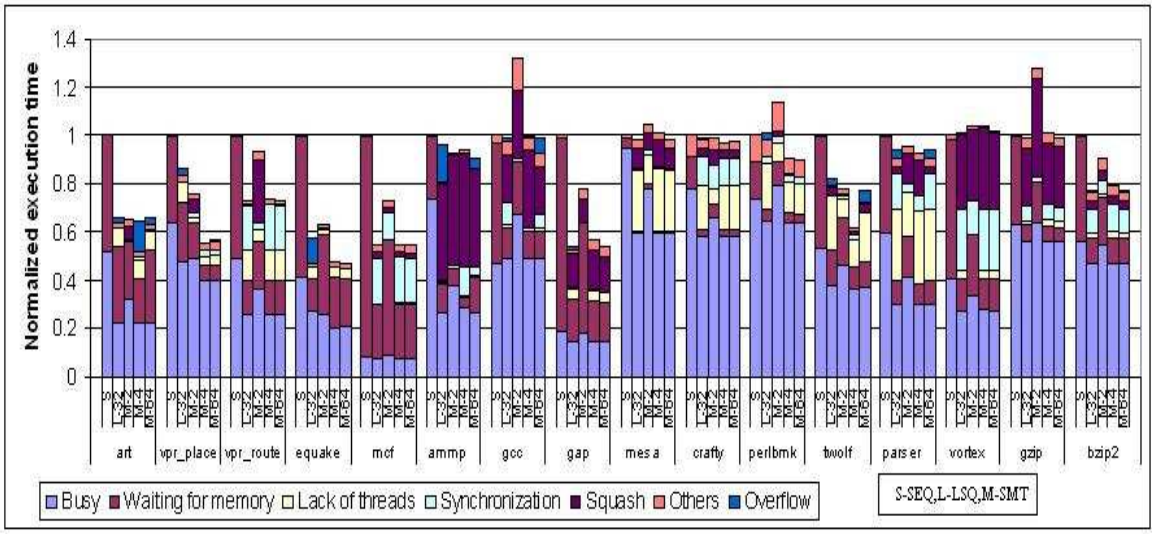
(b) Speedup for SPEC 2006 benchmarks.

Figure 4.6: Speedup of LSQ-32, SMT-2, SMT-4 and LSQ-64 configurations over SEQ.

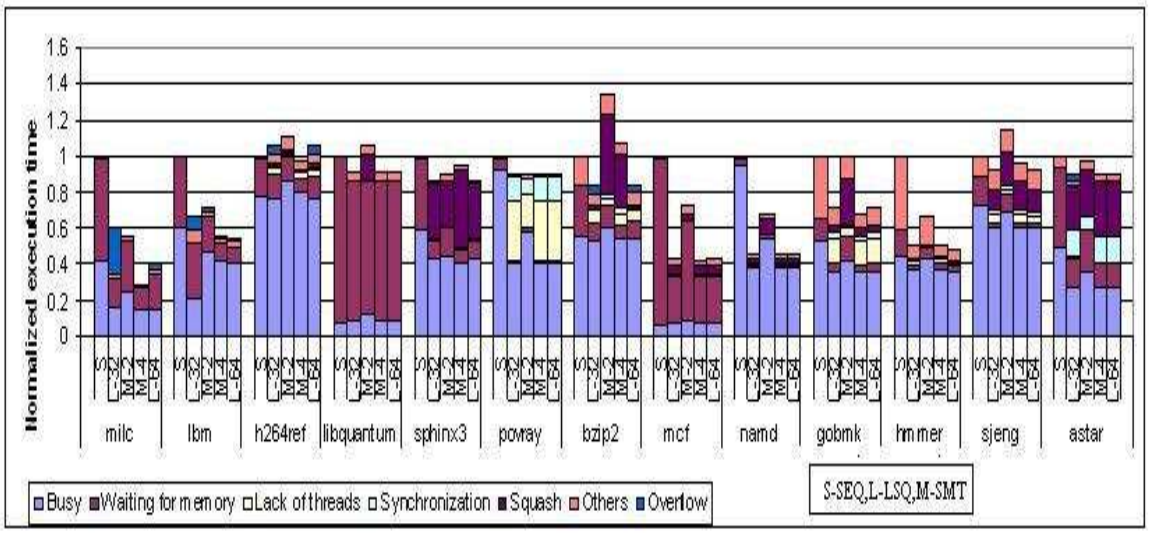
in all the benchmarks.

To better understand the performance and understand the merits of each of the architecture we present the execution time breakdown for parallel region execution in Figure 4.7(a) for SPEC 2000 and Figure 4.7(b) for SPEC 2006 benchmarks. In Figure 4.7(b) each bar is divided into six segments: *Busy* represents the amount of time spent in executing useful instructions and the delay due to lack of instruction level parallelism inside each thread; *Lack of threads* represents the amount of time wasted due to the lack of parallel threads (probably due to low iteration count in a loop); *Synchronization* represents the amount of time spent in synchronizing frequently occurring memory dependences and register dependences; *Cache misses* represents the amount of time the processor stalled due to cache misses; *Squash* represents the amount of time wasted executing instruction that are eventually thrown away due to failed speculation; *Overflow* corresponds to the amount of time wasted due to speculative state overflow and *Other* corresponds to everything else. In particular, it includes time wasted due to instruction cache misses, branch mis-predictions and load imbalance between consecutive threads.

The SMT-2 architecture which uses only two threads performs worse than other architectures in most benchmarks due to its reduced ability to exploit parallelism. For example in MILC SMT-4 is almost 2 times faster than the SMT-2. Similar slowdown due to reduced parallelism is significant in benchmarks like LBM, NAMD, MCF, HMMER, EQUAKE AND GCC. Also we can see that the SMT-2 architecture has more wastage due to thread violations than other architectures in benchmarks ART,VPR_ROUTE,VPR_PLACE, PERLBMK, GZIP, LIBQUANTUM AND GOBMK. These additional violations are caused due to *false violations* as the SMT-2 uses just



(a) Breakdown for SPEC 2000 benchmarks.



(b) Breakdown for SPEC 2006 benchmarks.

Figure 4.7: Execution time breakdown for parallel region execution of LSQ-32, SMT-2, SMT-4 and LSQ-64 configurations normalized to the SEQ configuration.

one SL bit for the entire cache block. If the non-speculative thread writes to the same cache block which was read by the speculative thread, the speculative thread is restarted even if the speculative thread had read from a different word in the same cache block than the location written by the non-speculative thread.

We overcome these limitations in the SMT-4 architecture which uses 4 threads as in the LSQ-32 configuration and also it uses additional SL bits per cache block to do fine-grained dependence checking and thus avoid *false violations*. In LSQ-32 configuration all the speculative state is buffered in LSQs of only 32 entries in size. When the LSQ is full, the thread is stalled till it becomes non-speculative. This stall due to such speculative overflow is shown in *Overflow* bars in Figure 4.7. We can see this effect is significant in benchmarks ART VPR_PLACE, EQUAKE, AMMP, TWOLF, PARSER, MILC, LBM AND H264REF. When compared to LSQ-32, the SMT-4 configuration uses the larger L1 data cache to buffer its results as seen in Section 4.4. Due to this we don't see overflow in SMT-4 in most benchmarks. In addition to increased stalling, the LSQ-32 also looses performance due to secondary effects that occur because of stalling. For example in benchmark VPR_PLACE the LSQ-32 suffers from increased delay due to cache misses than the SMT-4 architecture. These secondary effects cause more slowdown in VPR_PLACE than the actual stalling due to overflow. In benchmark ART which has very large threads (on the order of 100,000 instructions) SMT-4 also overflows.

In benchmark BZIP2 in SPEC 2006, the SMT-4 performs worse than LSQ-32 configuration. Here the LSQ-32 (also LSQ-64) suffer from overflow and the speculative threads are stalled. In the SMT-4 configuration though the threads are not stalled, they eventually are squashed due to

dependence violations. Due to increased violations in SMT-4 it performs worse than LSQ-32 configuration. Similar effect was also seen in benchmark ASTAR.

When we consider LSQ-64 configuration which has larger LSQs, the overflow effect is reduced leading to better performance. In almost all benchmarks, the LSQ-64 configuration matches the performance of SMT-4 configuration leading to equal overall performance in both SPEC 2000 and SPEC 2006 benchmarks.

We can see that even though the LSQ-32 architecture suffers from speculative state overflow in many benchmarks, the overall improvement due to the SMT-4 architecture is only 4.5% over all benchmarks. This is because many benchmarks do not suffer from overflow due to their smaller thread size. To highlight the effectiveness of our SMT-4 architecture, we show the performance of only the benchmarks which suffer from speculative state overflow in Figure 4.8.

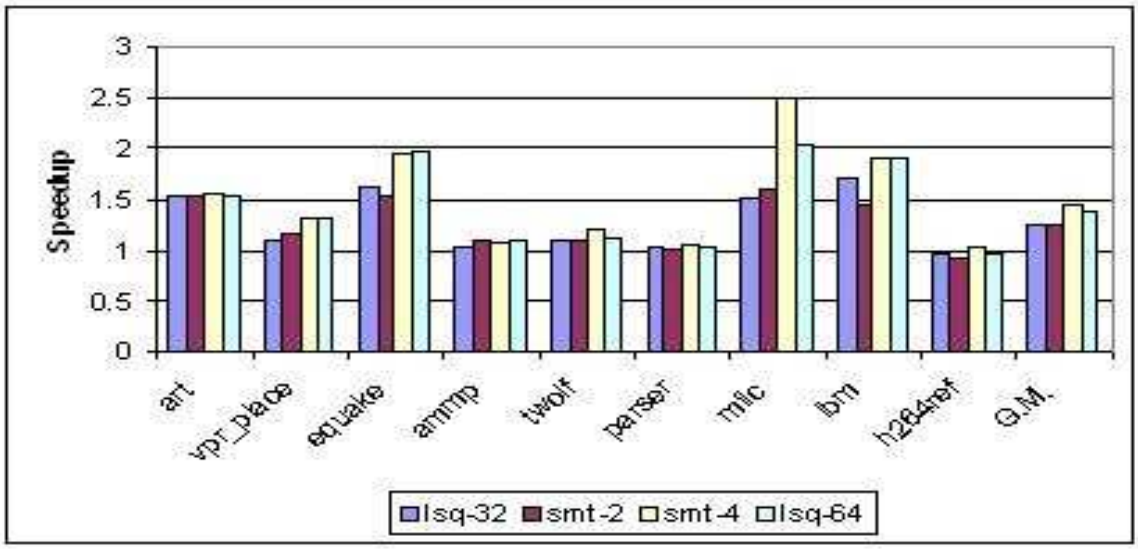


Figure 4.8: Speedup of LSQ-32,SMT-2,SMT-4 and LSQ-64 architectures over SEQ for selected benchmarks with *overflow* problem

From Figure 4.8. we can see that the SMT-4 configuration performs about 19% better than the LSQ-32 configuration and about 5% better than the LSQ-64 configuration. Even the SMT-2 configuration performs about 1% better than the LSQ-32 configuration due to reduced overflow in SMT-2.

4.6 Conclusions

In this chapter we proposed novel cache-based schemes to support TLS in SMT processors. We showed how our cache-based schemes can support TLS without using large associative structures like LSQs used in previous approaches. Also due to larger size of L1 data cache, we showed that our cache based design can support larger threads without suffering from stall due to speculative state overflow. Our novel two-thread scheme requires only few bits to be added to each cache line and with this simple modification we can achieve about 10% speedup over the superscalar processors. Our four-thread scheme with slightly more complex hardware outperforms LSQ based SMT design by about 4.5%. It outperforms LSQ based design by 19% if we consider only selected benchmarks which suffer speculative state overflow. In the next chapter we will study the efficiency of our cache based design and compare it with existing CMP based TLS designs.

Chapter 5

Performance/Power/Thermal Comparison

Although there have been numerous studies on the performance aspects of CMP based TLS (CMP-TLS) and SMT based TLS (SMT-TLS) architectures [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] , to the best of our knowledge there has not been a detailed comparative study on their *performance, power and thermal effects* when compared to the superscalar architecture *under the constraint of equal chip area*. In this chapter we present a detailed comparison of SMT and CMP based TLS architectures in terms performance, power consumption, energy efficiency and thermal behavior.

The power and performance characteristics of both the SMT and the CMP architectures have been studied extensively with various workloads: Lo *et. al* [31] show that SMT achieves better speedup for explicitly parallel workloads (from SPLASH-2); Kaxiras *et. al* [32] conclude

that SMT is more power efficient. Sasanka *et. al* show that CMP is more power efficient for multimedia workload and Burns *et. al* [35] show CMP to have higher throughput. However, it is difficult, if not impossible, to determine which architecture is more efficient for supporting TLS based on previous work, since the characteristics of TLS workload is fundamentally different from the studied workloads in the following ways:

Available parallelism vary during execution: Unlike multi-threaded workloads, the amount of parallelism that can be exploited by TLS may vary during execution of a program. In particular, different loops can have different amounts of parallelism; for significant portion of execution, efficient speculation threads cannot be extracted.

When available speculative threads are insufficient for utilizing all hardware threads, some of the cores will idle in a CMP-based architectures, while resources can be dynamically reallocated to exploit instruction level parallelism in SMT.

Resource competition and sharing between speculative and non-speculative threads: In SMT, the non-speculative thread competes and shares resources with speculative threads. Such competition can degrade the performance of the non-speculative thread. On the other hand, resource sharing can also benefit the performance of TLS. E.g., a speculative thread fetches data into the cache during it execution, even if the speculative thread is eventually squashed, the data it brought into the cache can potentially be used by the non-speculative thread or other speculative threads. Both SMT and CMP architecture benefit from the shared L2 cache, while the SMT gets extra benefit from its shared L1 cache.

Power consumption due to speculation failure: When data dependence is violated, the thread that contains the consumer of the dependence must be re-executed. Such re-executions can cause additional power consumption.

To conduct a detailed study comparing the two architectures, and to understand the relative merits of each architecture, we must identify two architectures with the same *cost*. In this chapter, we choose to compare two architectures with the same die area. A wide spectrum of design choices and trade-offs are studied using commonly used simulation techniques.

The rest of the chapter is organized as follows: Section 5.1 describes the related work. Section 5.2 considers various trade-offs and configures the three architectures, superscalar, SMT and CMP, with equal die area; Section 5.3 evaluates the performance, power consumption and energy-delay-product of each architecture under TLS workload; Section 5.4 studies the sensitivity of these results with several key architectural parameters; Section 5.5 presents the thermal effects of the TLS-workload on the three architectures; and in Section 5.6 we present our conclusions.

5.1 Related work

While the discussions on TLS performance have mostly been under the context of CMP [20, 21, 25], SMT processors can also be extended to support TLS [30, 73]. However, given the characteristics of TLS workload described earlier, it is not clear which architecture can achieve a higher performance and a better power efficiency while creating less thermal stress.

Renau *et. al* [43] compared the power efficiency of a CMP processor with TLS support against an equal-area, wide-issue superscalar processor. They concluded that the CMP processor with TLS support can be more power efficient on general-purpose applications. Their selection of equal-area configurations is based on a rough assumption that a 6-issue superscalar has the same area as a 4-core 3-issue CMP. In this chapter we conduct a detailed study of area overhead to identify equal area configurations. Also we include SMT based TLS in our comparison. Warg *et. al* [74], compared speedup of SMT and CMP based TLS architectures using simple assumptions to choose the configurations. In this chapter, we study several equal area configurations based on detailed area estimation. Also we present a detailed comparison which includes performance, power and thermal effects.

Numerous studies have compared the SMT and CMP performance and power efficiency under different workloads. On parallel programs [31] and mobile workloads [32], SMT processors outperform CMP processors. However, on multimedia workloads, CMP is more efficient [33]. In the context of multi-program workload, Li *et. al* [34] found that SMT is more efficient for memory-bound applications while CMP is more efficient for CPU-bound applications; Burns *e*t. al [35] found that SMT can achieve a better single thread performance, but CMP can achieve a higher throughput.

In terms of thermal effects of CMP and SMT processors, Donald *et. al* [75] found that SMT produces more thermal stress than CMP; while Li *et. al* [34] show that the two architectures have similar peak operating temperatures but SMT processors have more localized heating. In contrast to these studies which used multi-programmed workloads, we use TLS workloads to

study the thermal behavior.

Recently, there are many studies that explore the design space for CMP processors both using multi-programmed workloads [76] and parallel workloads [77]. In this chapter, we study the design space for TLS workloads.

5.2 Processor configurations

For fair power and performance comparisons among superscalar, CMP-TLS and SMT-TLS architectures, three different processor configurations are constructed with equal chip area. We use a detailed area estimation tool presented in [78]. The tool, which originally targets only SimpleScalar-based architectures, is extended to estimate area of SMT and CMP architectures.

However, even for a fixed chip area, many processor configurations are possible by varying the size of the cores and the caches; and it is not possible to exhaustively evaluate the entire design space. In this section, we describe how equal-area processor configurations are selected for fair comparisons in this study.

5.2.1 Superscalar configuration

The baseline of this study is a SimpleScalar-based superscalar architecture. The architectural parameter of this processor can be found in Table 5.1: the die area occupied by each component of this processor can be found in Table 5.2, estimated using the die-area estimation tool [78] (assuming 70nm technology). We refer to this architecture as the *SEQ architecture*.

Table 5.1: Architectural parameters for the superscalar (SEQ) configuration and the SMT configurations with 2 and 4 threads

Parameter	superscalar(SEQ)	SMT-4	SMT-2
Fetch/Decode/Issue/Retire Width	12/12/8/8	12/12/8/8	12/12/8/8
Integer units	8 units / 1 cycle latency	7 units	7 units
Floating point units	5 units / 12 cycle latency	4 units	5 units
Memory ports	2Read, 1Write ports	2R,1W	2R and 1W
Register Update Unit (ROB,issue queue)	256 entries	196	234
LSQ size	128 entries	96	110
L1I Cache	64K, 4 way 32B	64K, 4 way 32B	64K, 4 way 32B
L1D Cache	64K, 4 way 32B	64K, 4 way 32B	64K, 4 way 32B
	Common to superscalar and SMT		
Cache Latency	L1 1 cycle, L2 18 cycles		
Memory latency	150 cycles for 1st chunk, 18 cycles subsequent chunks		
Unified L2	2MB, 8 way associative, 64B blocksize		
Branch predictor	2K Pattern history table (PHT), 2K Branch target buffer (BTB)		
Branch mis-prediction penalty	6 cycles		
Physical registers per thread	128 Integer, 128 Floating point and 64 predicate registers		
Thread overhead	5 cycles fork, 5 cycles commit and 1 cycle inter-thread communication		

5.2.2 SMT configuration

The SMT architecture is extended to support TLS based on the technique described in chapter 4. When compared to the superscalar architecture, the SMT architecture incurs additional overhead to support threads. To configure a SMT processor with an equal area, we need to reduce the complexity of the processor to compensate for the area increase due to threads. First, we show the estimated overhead of different components and then we show how we reduce the processor complexity to get the desired equal-area configuration.

Table 5.2: Die area estimation for (1) superscalar (SEQ), (2) SMT processor with same complexity as SEQ and (3) SMT processor with reduced complexity occupying an equal area.

Hardware structures	Area effect due to			Area in $M\lambda^2$		
	Issue width (d)	SMT Threads(t)	Function units(f)	superscalar	SMT with overhead	adjusted SMT
Function units						
Integer units	None	None	O(f)	1,057.96	1,057.96	925.71
Floating point units	None	None	O(f)	1,436.73	1,436.73	1,149.39
Load store units	None	None	O(f)	450.00	450.00	450.00
				2,944.69	2,944.69	2,525.10
Pipeline logic:						
Fetch unit	O(d)	O(t)	None	390.00	487.50	487.50
Decode unit (dispatch)	O(d)	20% overhead	None	360.00	396.00	396.00
Issue (scheduler)	O(d)	None	None	320.00	352.00	352.00
Write back unit	None	None	O(f)	320.00	352.00	308.00
Commit unit	O(d)	20% overhead	None	176.00	202.40	202.40
				1,566.00	1,789.90	1,745.90
Register File	O(min (f,d))^2	O(t)	O(min (f,d))^2	1,111.46	4,445.83	4,128.83
LSQ	None	None	O(f^2)	1,446.30	1,473.33	954.99
RUU	None	None	O(f^2)	15,215.25	16,911.31	11,515.58
BTB, ALAT, IFQ				740.00	2113.00	1,320.10
Caches						
TLBS	No change	Extra port	No change	105.46	116.00	116.00
Level 1 i-cache	No change	Extra port	No change	1426.72	1956.64	1,956.64
Level 1 d-cache	No change	Extra TLS bits	O(f)	1956.64	2,257.07	2,257.07
Level 2 cache	No change	No change	No change	41397.78	41397.78	
Total Area without L2				26,512.04	34,007.61	26,520.22
Total Area				67,909.83	75,405.39	67,918.00
Total chip area in mm^2 *for 70nm technology*				83.2	92.4	83.2

SMT overhead due to threads

Table 5.2 shows the area overhead of the SMT architecture with the same configuration as a superscalar architecture. Estimated area overhead for different components are discussed below:

Register Update Unit (RUU): The RUU occupied 57% of the core area and is the central hardware support for out-of-order execution. In SMT, the RUU is shared by several threads, thus a thread identifier is needed to distinguish the entries. This leads to about 10% increase in area.

Register file: Each thread needs its own register file. This leads to 4 times increase in the area of register file.

Fetch logic: The area cost for the instruction fetching stage is increased by approximately 25% to support the *icount* fetch policy. The branch predictor, I-cache and I-TLB need extra port to support fetching from multiple threads. The return address stack is replicated for each thread.

Data cache: The area for the first level data cache increases approximately 15% for storing the extra bits required for supporting TLS [73].

Out-of-order logic: If we consider the different components that constitute the out-of-order like the RUU, LSQ, create vector and pipeline stages, the overhead due to 4 threads is 11% in our estimation. In [35] it is estimated to be about 68% mainly due to the duplication of

larger remap-tables for supporting 8 threads.

The overall area cost for supporting a SMT processor of the same configuration as superscalar (SEQ) is approximately 28%. When taking into consideration the L2 cache, the area overhead of the entire chip is approximately 11%.

Configuring equal-area SMT processor

The complexity of the core can be reduced by changing several parameters, but our main target is the RUU(Register Update Unit) since it occupies significant die area (about 57% in SEQ). However, simply reducing the number of RUU and LSQ (Load Store Queue) entries while holding other parameters constant implies a 40% reduction in RUU entries. This approach clearly creates a performance bottleneck, and thus produces a sub-optimal design. RUU requires many ports, since it is the central structure accessed by almost all pipeline stages. By reducing the number of function units, we can reduce the number ports in RUU, in turn, reduce the area cost of RUU. Thus, we reduce both the number of function units and the number of RUU and LSQ entries to achieve the desired area cost. The exact configuration chosen for SMT configuration is shown in Table 5.1. In Table 5.2, the area of each component in this equal area SMT configuration is shown.

To study the impact of the reduction in the number of TLS threads, we include a configuration called SMT-2 which supports 2 threads (equal area as SEQ and SMT-4). Reducing the number of threads reduced the area overhead by about 13% (without L2), and we follow a similar strategy of reducing the number of RUU entries and function units to compensate for the

Table 5.3: Architecture Parameters for the CMP configurations with: 4 cores + 2MB L2 cache; 2 cores + 2MB L2 cache; 4 cores + 1MB L2 cache;

Parameter	CMP-4-2MB	CMP-2-2MB	CMP-4-1MB
Fetch/Decode/Issue/Retire Width	6/6/4/4	9/9/6/6	9/9/6/6
Integer units	4 units	6 units	6 units
Floating point units	2 units	4 units	3 units
Register Update Unit	106 entries	148 entries	122 entries
LSQ size	44 entries	74 entries	64 entries
L1 D-cache size	16K	32K	32K
L1 I-cache size	16K	32K	32K
Unified L2	2MB, 8 way associative, 64B blocksize		1MB, 8 way associative, 64B blocksize
Branch predictor	1K Pattern history table (PHT), 1K Branch target buffer (BTB)		

SMT overhead. The configuration for SMT-2 is shown in Table 5.1.

5.2.3 CMP configurations

In choosing the area-equivalent CMP configurations we have two design choices. One way is to hold the L2 size the same as in SEQ and allocate less area for each core, so the total area for the multiple cores is the same as that of the superscalar core (as in [35]). The second choice is to reduce L2 cache size and use the area for the cores (as in [34]). Alternatively, we could reduce the number of cores supported, which will allow us to use larger cores. To cover all these design choices, three different configurations of CMP architecture are studied - CMP-4-2MB(CMP-4cores-2MB L2 cache), CMP-4-1MB, and CMP-2-2MB.

The specifics of each of configuration are shown in Table 5.3. We estimated the area of each configuration and made sure they have the same area (shown in Table 5.4).

Table 5.4: Die area estimation for CMP variants.

Hardware structures	CMP-4-2MB	CMP-4-1MB	CMP-2-2MB
	area (mm^2)	area (mm^2)	area (mm^2)
Function units			
Integer units	0.648	0.972	0.972
Floating point units	0.704	1.056	1.408
Load Store units	0.367	0.551	0.551
	1.719	2.579	2.931
Pipeline logic			
Fetch unit	0.239	0.358	0.358
Decode unit	0.220	0.330	0.330
Issue unit	0.196	0.294	0.294
Writeback unit	0.196	0.294	0.319
Commit unit	0.108	0.161	0.161
Caches			
TLBs	0.104	0.129	0.129
L1 I-cache	0.439	0.877	0.877
L1 D-cache	0.503	1.362	1.330
Register file	0.414	0.802	0.874
RUU	1.943	4.73	6.1
LSQ	0.194	0.574	0.710
Misc	0.344	0.463	0.458
Core Size	6.6	12.95	14.76
Bus area		5.95	2.975
L2 cache	50.71	25.35	50.71
Chip size	83.2	83.2	83.2

5.3 Performance and power comparisons

We compare the three different architectures - CMP-based TLS, SMT-based TLS and superscalar in terms of performance in Section 5.3.1. In Section 5.3.2, we compare their power consumption, and in Section 5.3.3, we use *energy-delay product* (ED) and *energy-delay-squared product* (ED^2) to compare energy efficiency. We use the evaluation framework describe in chapter 2 to compare the different architectures.

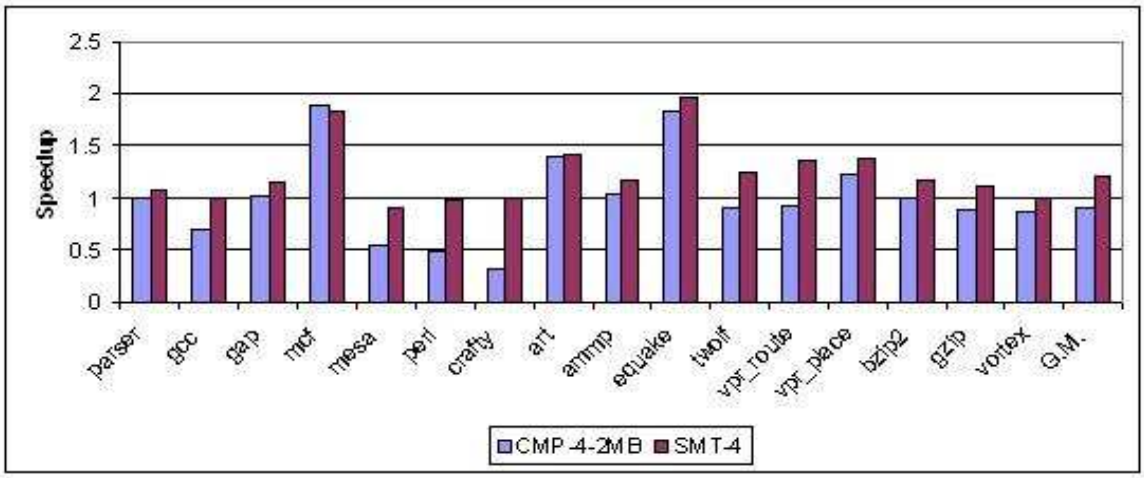
5.3.1 Performance

Fig. 5.1 shows the speedup of the entire benchmark suite using superscalar (SEQ) performance as the base and Fig. 5.2 shows the breakdown of execution time when executing loops selected by the compiler. In this section, we only show the TLS configurations: CMP-4-2MB and SMT-4. We will discuss other possible configurations in Section 5.4.

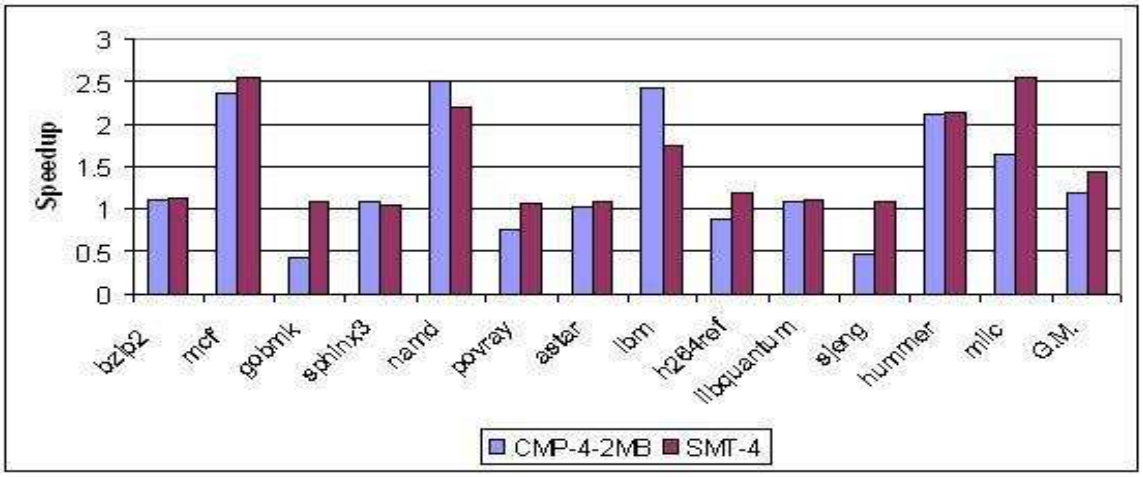
For SPEC 2000 benchmarks, the CMP-4-2MB slows down in *perlbmk*, *gcc*, *twolf*, *mesa*, *gzip*, *vortex*, *vpr_route* and *crafty*, leading to a *geometric mean* (GM) slowdown of 8% when compared to SEQ. Due to its dynamic sharing of resources, SMT-4 is able to extract good performance even in benchmarks with limited parallelism except in *mesa* and *perlbmk*, leading to about 21% speedup over SEQ. In SPEC 2006 benchmarks, the CMP-4-2MB achieves speedup in most benchmarks except in *gobmk*, *povray*, *h264ref* and *sjeng*, leading to a *geometric mean* (GM) speedup of 18% when compared to SEQ. SMT-4 gains speedup in all benchmarks and achieves an overall performance of 39% better than SEQ.

Each benchmark benefits from specific architecture depending on its characteristics. A comparison of the impact of different benchmark characteristics on the TLS performance in CMP and SMT architectures is presented in Table 5.5.

Large sequential non-parallelized code regions: The CMP-4-2MB slows down about 8% compared to SEQ in SPEC 2000 benchmarks when we consider the entire benchmark (Fig. 5.1), while it achieved about 13% speedup if we consider only the parallel regions (in Fig. 5.2). Similarly it achieves about 40% speedup if we consider the parallel regions in SPEC 2006, but the speedup is only 18% if we consider the entire benchmark. Many of the benchmarks

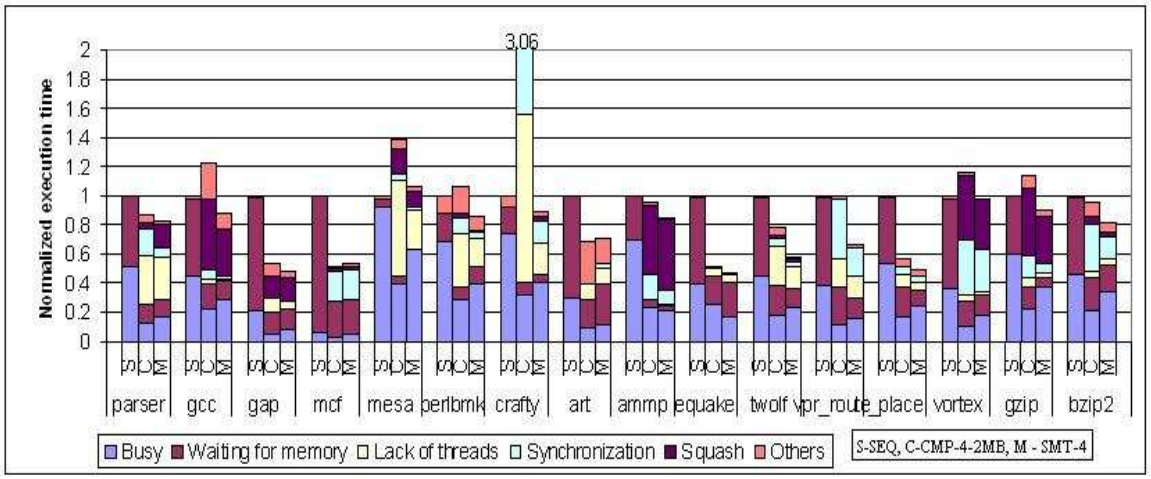


(a) Speedup for SPEC 2000 benchmarks.

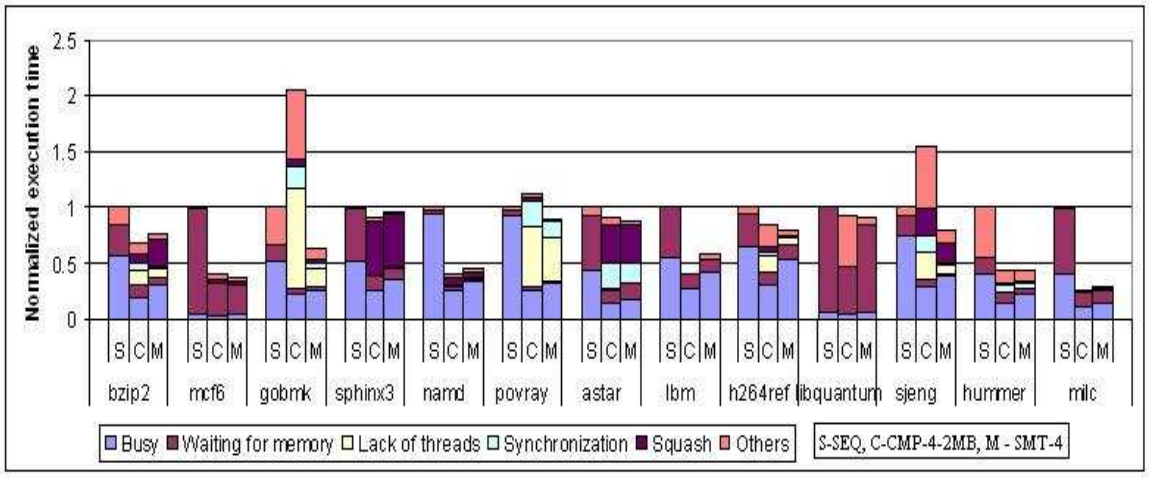


(b) Speedup for SPEC 2006 benchmarks.

Figure 5.1: Speedup of CMP-4-2MB and SMT-4 configurations over SEQ.



(a) Breakdown for SPEC 2000 benchmarks.



(b) Breakdown for SPEC 2006 benchmarks.

Figure 5.2: Execution time breakdown for parallel region execution of CMP-4-2MB and SMT-4 configurations normalized to the SEQ configuration.

Table 5.5: Comparison of the impact of benchmark behaviors on the performance of SMT-TLS vs CMP-TLS.

Benchmark characteristics	Impact on CMP	Impact on SMT	Reasons
Large sequential regions	X	✓	SMT could use all resources to extract ILP inside sequential regions.
Low TLP inside parallel regions	X	✓	SMT effectively uses all its resources while many cores in CMP could be idle
High cache miss rates	✓	✓	Both can hide memory latency and speculative threads can prefetch data. SMT has more advantage due to shared L1.
Threads with a large working set	✓	X	SMT L1 cache overflows more often as it is shared by all threads.
Frequent mis-speculations	X	X	Mis-speculations wastes resources and affects non-speculative thread performance in SMT.

considered have significant sequential (non-parallelized) regions which suffer poor performance on CMP-4-2MB due to its static partitioning of resources. The *perlbmk* shows more than 50% slowdown for CMP-4-2MB configuration. The coverage of sequential regions in *perlbmk* is about 75%. Due to this very low parallel-region coverage, we see a huge decrease in overall performance for *perlbmk*. In benchmark *twolf*, the CMP performs about 29% better than SEQ when we consider parallel regions. But when we consider the entire benchmark, the CMP performs about 8% worse than SEQ due to only 47% coverage of parallelized regions. Similarly in *h264ref* CMP-4-2MB achieves about 17% speedup inside parallel regions but slows down by about 10% when we consider the entire benchmark. Similarly, *crafty, vpr_place, gobmk, sjeng and bzip2(SPEC 2006)* suffer from poor sequential region performance.

On the other hand, the SMT configuration was able to dynamically reallocate its resources to exploit ILP when executing in sequential regions. Even though there is a slight slowdown in some benchmarks for SMT, the impact is much less when compared to CMP. For example, in *h264ref* SMT-4 performs 19% better than SEQ while CMP-4-2MB slows down by about 10%,

inspite of both achieving similar speedup inside parallel regions. Overall, SMT-4 performs about 44% better than SEQ if we consider only the parallel regions in SPEC 2000 benchmarks while its performance reduces to 21% when we consider the entire benchmark. In SPEC 2006 benchmarks the parallel region speedup is about 58% while the overall benchmark performance is 44%.

Low TLS parallelism inside parallelized regions: Even in benchmarks with high parallel region coverage, the CMP-4-2MB could slowdown when compared to SEQ due to limited parallelism inside the parallelized regions. For example, in benchmark *povray*, the parallel region coverage is 63%, but the loops selected have very poor iteation count leading to many threads being idle (indicated as *lack of threads*). Due to the limited parallelism available, the CMP did not get good performance, while SMT due to its dynamic resource allocation, uses the resources to extract ILP within the threads, resulting in a better performance than CMP. Here the SMT-4 is able to dynamically re-allocate its resources to exploit instruction-level parallelism inside the available threads leading to about 5% speedup. Similarly in *crafty* the CMP-4-2MB suffers due to synchronization and poor iteration count. Similar effect can be seen in *gcc, mesa, perlbmk, crafty, gzip, vortex and vpr_route*. In all these cases the SMT-4 is able to dynamically re-allocate its resources to exploit instruction-level parallelism in the remaining threads.

Large number of cache misses: In benchmark *equake* and in *mcf*, the SEQ configuration spends most of the execution time waiting for memory due to a large number of cache misses. Both CMP-4-2MB and SMT-4 are able to better hide the memory latency through sharing of the common working set. Such sharing of the working set allows some data needed by one

thread to be *pre-fetched* by another thread. The benchmarks *equake* and *mcf* have excellent TLS parallelism, consequently, there are very few squashes. Due to the combined effect of parallelism and prefetching, both CMP-4-2MB and SMT-4 achieve good performance. Similarly, benchmarks *milc*, *libquantum*, *twolf* and *vpr_place* gain from good TLS parallelism and cache prefetching leading to performance gain for both SMT and CMP.

In SMT, both L1 cache and L2 cache are shared by all the threads, leading to better prefetching when compared to CMP where the threads share only the L2 cache. In *twolf, vpr_place and vpr_route*, SMT-4 performs better than CMP-4-2MB due to prefetching effect in L1 cache.

Effect of frequent squashes: Frequent dependence violations hurts the performance of both SMT-4 and CMP-4-2MB configurations as seen in benchmarks *gcc, vortex, gzip, mesa, sphinx3, astar and bzip2(SPEC 2006)*. But in the case of SMT-4, since the processor resources are shared, speculative threads compete for resources with the non-speculative thread. If the speculative work done is eventually wasted due to dependence violations we do not gain any thread-level parallelism while we also hurt the performance of non-speculative thread. Due to this there is a slowdown in SMT-4 when compared to CMP-4-2MB when there are frequent squashes as shown in Figure 5.2 for benchmarks *bzip2 and sphinx3* in SPEC 2006.

5.3.2 Power

To understand the power behavior of the two architectures, we compare the breakdown of dynamic power consumption in Fig. 5.3. The power consumption is normalized to the total power consumption of SEQ configuration. We used ideal clock gating (cc2) in the *Wattch* simulator to

get dynamic power consumption.

Dynamic power is proportional to $\alpha.C.V^2.f$, where α is the activity factor, C is the capacitance of the transistor, V the supply voltage, and f the frequency of the circuit. In our simulation, we kept V and f the same for all three configurations. So dynamic power differences among the three configurations are mainly due to the *a*ctivity factor or the *c*apacitance of the circuit.

Core complexity: The superscalar uses the most complex core and has the highest C value while SMT core is also complex. But the CMP configuration uses smaller cores and, hence, has a smaller C value than that in superscalar and SMT. The largest component of dynamic power, we call it the *window power*, combines the power consumption of function blocks related to out-of-order execution including RUU, LSQ, result bus, etc. The CMP configuration uses a smaller instruction window leading to lower window power consumption across all benchmarks. Similarly, it consumes less power in the cache since it uses a smaller cache than in other configurations as shown in Fig. 5.3.

Activity factor: SMT and CMP both execute the same parallel TLS code so their activity factor is very similar. However, SEQ runs the sequential code which does not have any special TLS instructions, leading to a smaller activity factor than SMT and CMP. Another factor which affects the activity is the amount of speculation. Both SMT and CMP suffer from frequent dependence violations, but the power wasted due to squashes in SMT is higher due to its higher complexity. This effect can be seen in benchmarks *ammp, mesa, gzip, vortex, astar, sphinx3 and bzip2(SPEC 2006)* The SEQ has a more complex core than both SMT and CMP, and thus

consumes higher power. But due to its lower activity factor its power consumption is lower than SMT.

Extra hardware: The TLS architectures have extra power overhead due to the extra hardware needed to implement TLS. The extra hardware used by SMT is minimal, but CMP uses a common bus to connect the cores. The power overhead due to this common bus is significant, and not present in SEQ and SMT configurations.

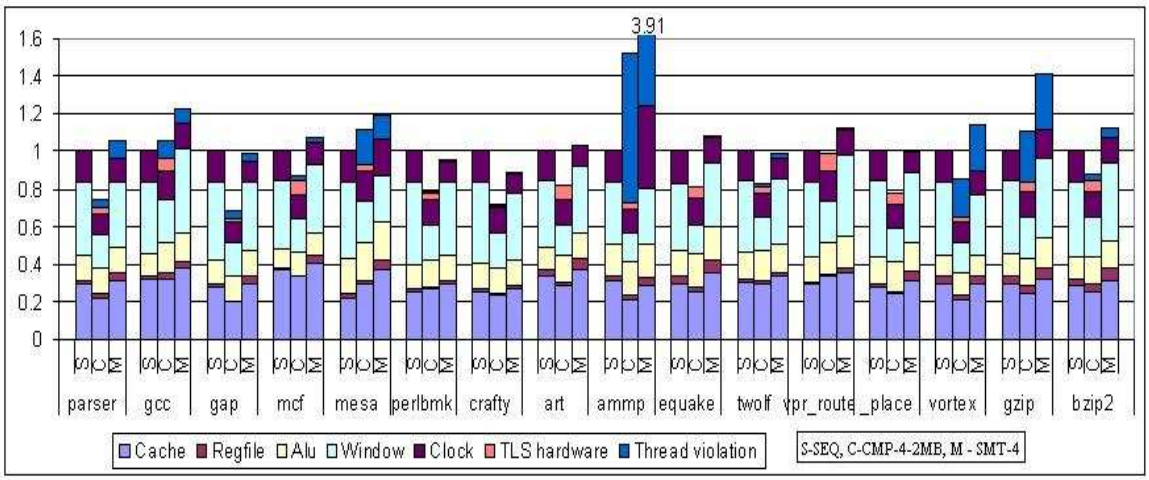
Overall, due to the combined effect of complex cores and speculative wastage, SMT on average consumes about 17% more dynamic power than SEQ for SPEC 2000 and 22% more dynamic power for SPEC 2006 benchmarks. CMP, due to its smaller cores, consumes about 11% less dynamic power than SEQ in SPEC 2000 and 3% lesser for SPEC 2006 benchmarks.

Total power: Total power consumption of the processor includes leakage/static power in addition to the dynamic power considered above. To get total power consumption, we use aggressive clock gating in *W*attch simulator (cc3).

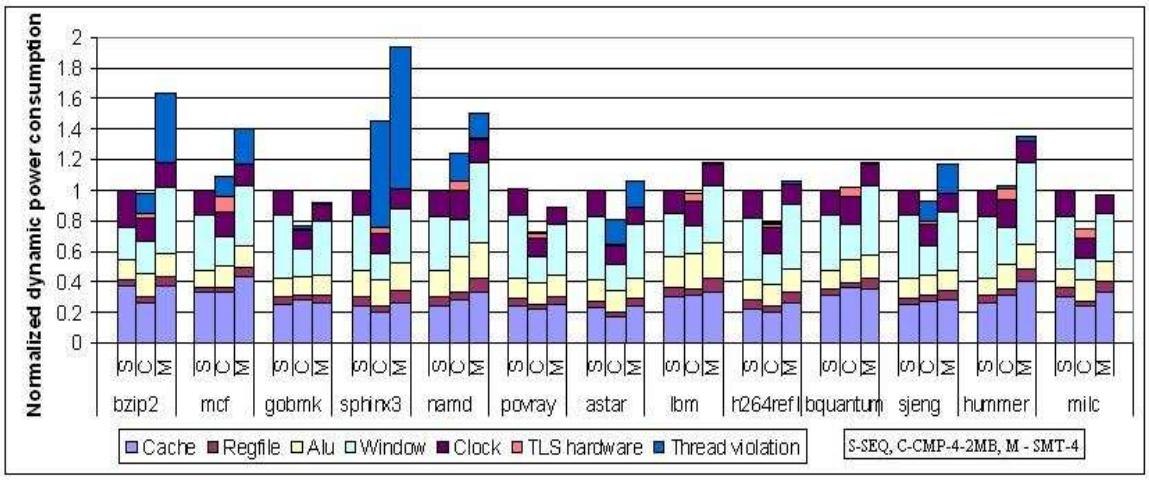
The static power consumption depends on the program execution time and on the number of components that have leakage power (i.e. number of transistors).

Both SMT-4 and CMP-4-2MB achieve good speedup in many benchmarks which leads to lesser static power than SEQ. Also SMT-4 consumes lesser static power than CMP-4-2MB due to its higher speedup.

The CMP-4-2MB configuration due to its lower complexity can pack more resources in the same chip area. For example, the CMP-4-2MB uses two times the number of function units, RUU entries, etc. Due to the use of a larger number of components, the CMP has more leakage



(a) Dynamic power consumption for SPEC 2000 benchmarks.



(b) Dynamic power consumption for SPEC 2006 benchmarks.

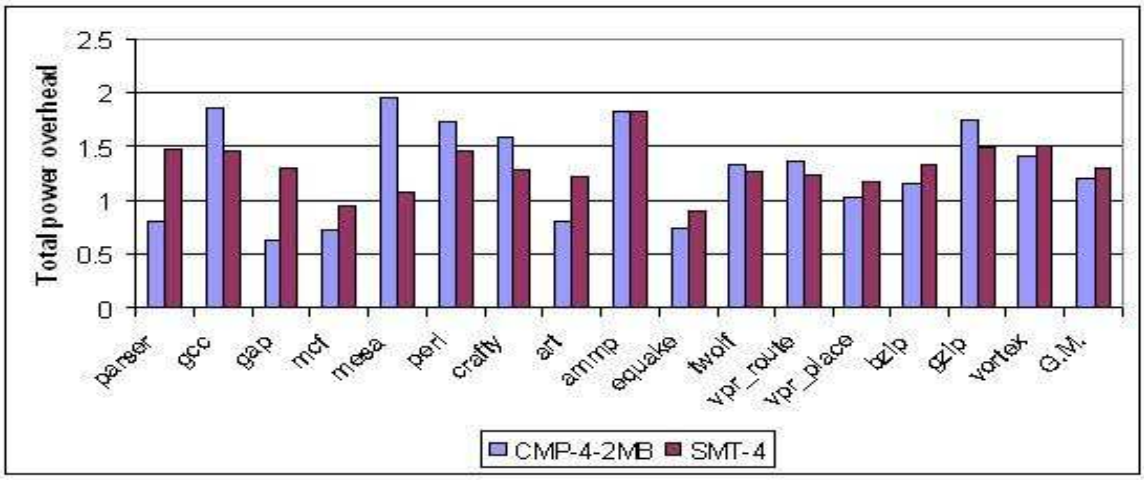
Figure 5.3: Normalized dynamic power consumption of CMP-4-2MB and SMT-4 configurations normalized to the power consumption of SEQ.

power than SMT. The SMT-4 also uses more resources when compared to SEQ. The register file is four times larger in SMT-4 when compared SEQ and also it uses additional bits in the L1 data cache. Due to additional resources both SMT-4 and CMP-4-2MB can consume additional leakage power when compared SEQ.

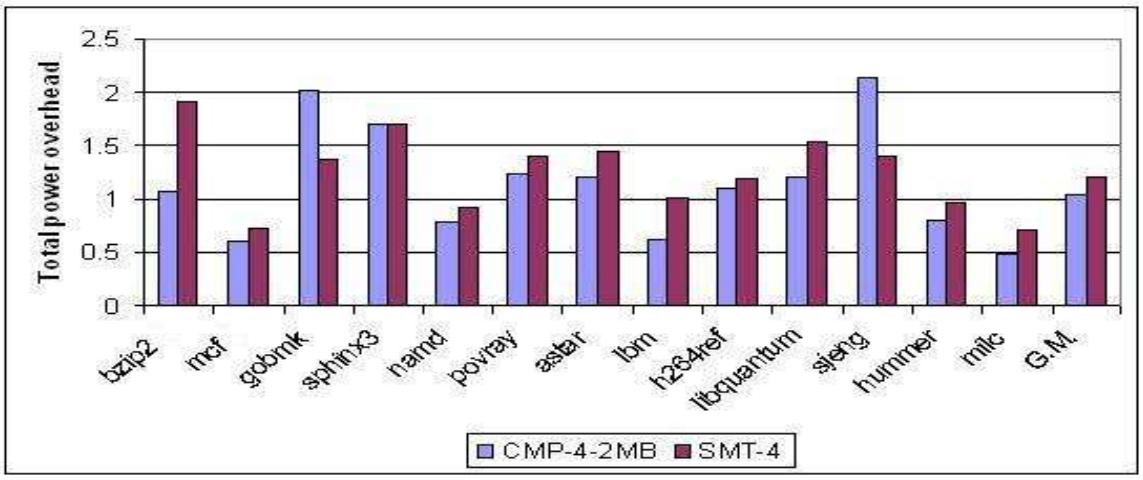
Fig. 5.4 shows the total power overhead for SMT and CMP over SEQ. The SMT-4 due to its good speedup in benchmarks *mcf, ammp, equake, sphinx3, namd, lbm, hummer, milc* is able to makeup for its increase in dynamic power. For example in *ammp*, the total power overhead of SMT is only 82% when compared to 291% overhead for dynamic power. Whereas in benchmarks such as *parser, gcc, gap, crafty, perlbmk, gobmk, povray, astar* where the speedup of SMT-4 is limited, the SMT-4 configuration suffers from more leakage due to its larger register file and other resources.

Similar to SMT-4, the CMP-4-2MB due to its good speedup in *mcf, equake, namd, lbm, hummer, milc* suffers from lower leakage power. But in other benchmarks due to its limited speedup and its larger resources, it suffers from increased leakage.

Overall, the CMP-4-2MB in spite of its lower dynamic power consumption suffers from 20% total power overhead when compared to SEQ due to increased static power caused by lower performance and larger resources. The SMT-4 due to its dynamic power overhead due to dependence violations and static power overhead due to its register file leads to about 30% power overhead. The CMP-4-2MB due to its higher speedup among SPEC 2006 benchmarks suffers only 4% overhead when compared SEQ. The SMT-4 configuration due to its higher



(a) Overhead for SPEC 2000 benchmarks.



(b) Overhead for SPEC 2006 benchmarks.

Figure 5.4: Total power overhead CMP-4-2MB and SMT-4 configurations over SEQ.

Table 5.6: Comparison of the impact of various factors on the power consumption of SMT-TLS vs CMP-TLS.

Different factors	Impact on		Reasons
	CMP	SMT	
Core complexity	✓	X	CMP with simpler cores consumes lesser dynamic power as seen in Fig. 5.3
Execution time	X	✓	SMT has lower execution time than SEQ leading to lower leakage. But CMP slows down in some benchmarks leading to more leakage.
Frequent squashes	X	X	Squashing leads to additional dynamic power in both SMT and CMP.
Number of transistors	X	✓	More transistors in CMP cause more leakage than in SMT.

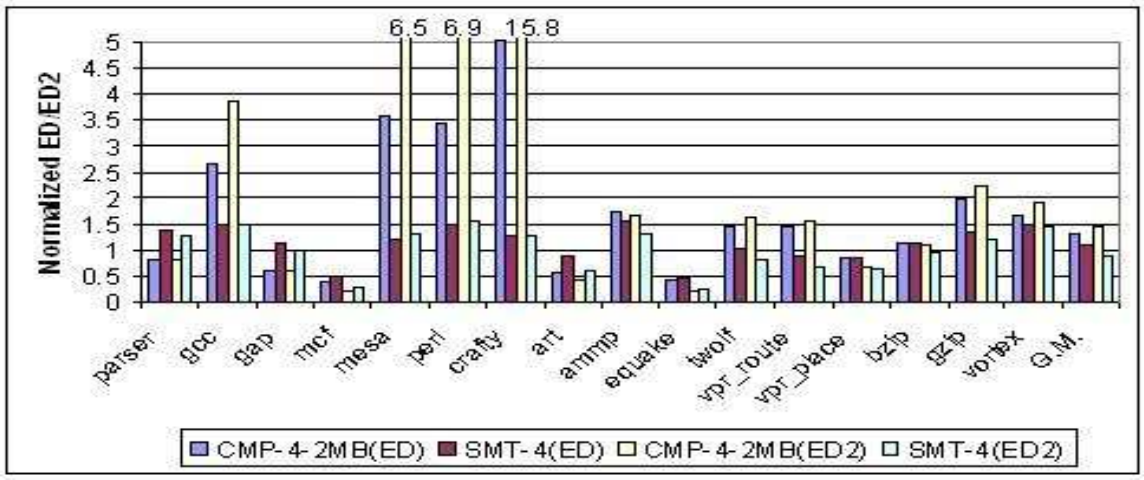
dynamic power overhead caused from higher number of mis-speculations in SPEC 2006 benchmarks leads to about 19% overhead when compared to SEQ. Overall all the benchmarks the CMP-4-2MB consumes 12% lesser power than the SMT-4 configuration.

A summary of how the various factors affect power consumption in SMT and CMP is presented in Table 5.6.

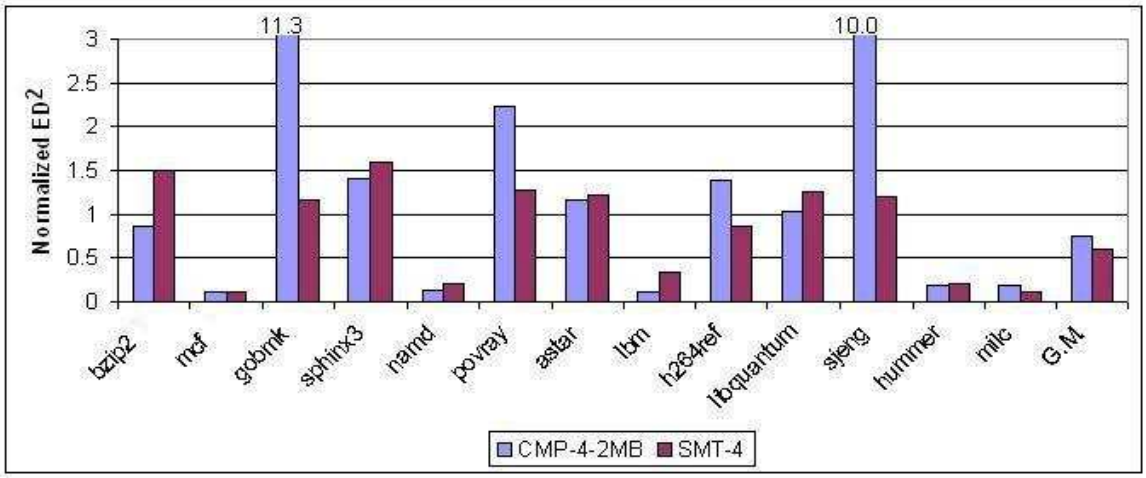
5.3.3 ED and ED^2

From the previous sections, we see that SMT and CMP have a very different behavior in power consumption and performance. To combine their effects we use *energy-delay product* (ED) and *energy-delay-squared product* (ED^2).

Fig. 5.5 shows the ED and ED^2 when we consider the entire program execution. As discussed before, the CMP-4-2MB configuration suffers from poor performance in SPEC 2000 benchmarks. Due to the poor performance the ED for CMP-4-2MB is about 24% worse than SMT-4 and 31% worse than SEQ. When we consider ED^2 the effect of poor performance further worsens leading to 55% worse ED^2 than SMT-4. The SMT-4 configuration due to its higher



(a) ED/ED^2 for SPEC 2000 benchmarks.



(b) ED/ED^2 for SPEC 2006 benchmarks.

Figure 5.5: ED/ED^2 of CMP-4-2MB and SMT-4 configurations normalized to SEQ.

power overhead suffers from 7% worser ED than SEQ. When we consider ED^2, it gains 11% better ED^2 when compared to SEQ.

When we consider SPEC 2006 benchmarks, where the CMP-4-2MB gains better performance and SMT-4 suffers from extra power overhead due to frequent squashes, the CMP-4-2MB is 12% better than SEQ in terms of ED and the SMT-4 is 17% better than SEQ, which is only 5% better than CMP-4-2MB. When we consider ED^2 the SMT-4's lead increases due to its better performance. Now the SMT-4 is 17% better than CMP-4-2MB.

From Fig. 5.5, we can see that though in most benchmarks SMT-4 does better than CMP-4-2MB, there are some benchmarks where CMP-4-2MB does better. In benchmarks with good parallelism like *mcf, art, equake, sphinx3, namd, lbm, libquantum and hummer*, the CMP-4-2MB does better. Also in benchmarks like *astar, bzip2('06), gap and parser* where the CMP-4-2MB has limited speedup and where the SMT-4 has increased power overhead, the CMP-4-2MB does better. In Fig. 5.6 we show the comparison of ED and ED^2 for SPEC 2000 and SPEC 2006 benchmarks classified in terms of their dependence behavior. In chapter 3 we classified the benchmarks into two classes depending on the number of inter-thread dependences: Class-A which has fewer inter-thread dependences and Class-B which has more number of inter-thread dependences. As we see in Fig. 5.6, for Class-A benchmarks which have better parallelism, the CMP-4-2MB beats SMT-4 in terms of ED by about 1%. Though, even among class-A benchmarks, *povray and mesa* suffer from limited speedup due to their limited iteration count and other factors. Due to this the SMT-4 beats CMP-4-2MB in terms of ED^2; though the difference now is only 9% when compared to 50% difference in Class-B benchmarks which

generally have limited parallelism due to inter-thread dependences.

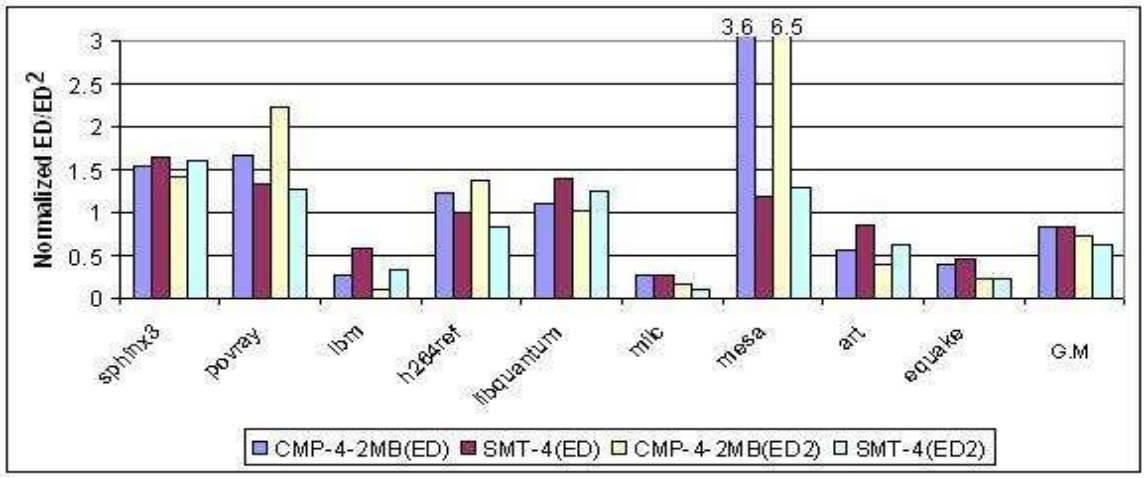
From the above discussion, it is clear that overall, the SMT-4 configuration is more efficient in extracting TLS parallelism than the CMP-4-2MB configuration. Though the CMP-4-2MB does better in benchmarks with more parallelism. In the next section, we consider different variations in the design space of CMP and SMT.

5.4 Alternative configurations

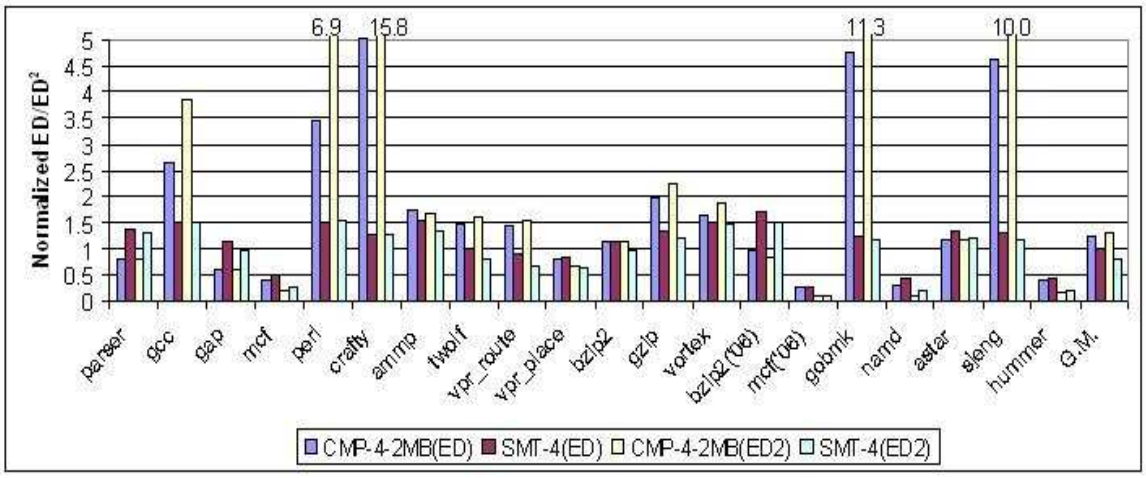
As we saw in previous section, the CMP based TLS performs worse than SMT based TLS due to its poor performance when executing in sequential regions. In this section, we study how the performance and power behavior change when we increase the core complexity to improve performance in sequential regions by varying key parameters such as the number of threads and L2 size.

Impact of the number of threads:

In Fig. 5.7 we compare the speedup of the 4-thread and 2-thread versions of both CMP and SMT architectures and in Fig. 5.8 we compare them in terms of ED^2. The CMP-2-2MB can support only two threads leading to poor parallel region performance but its larger cores helps to exploit instruction level parallelism inside sequential regions. In SPEC 2000 where the CMP-4-2MB suffers significant slowdown due to sequential regions, the CMP-2-2MB gains about 6% speedup due to its better sequential region performance. Though the CMP-2-2MB suffers from higher power overhead due to its complex core and it slows down in parallel regions, it gains better ED^2 of about 14% due to its better performance in benchmarks like *gcc, mesa, perlbmk,*

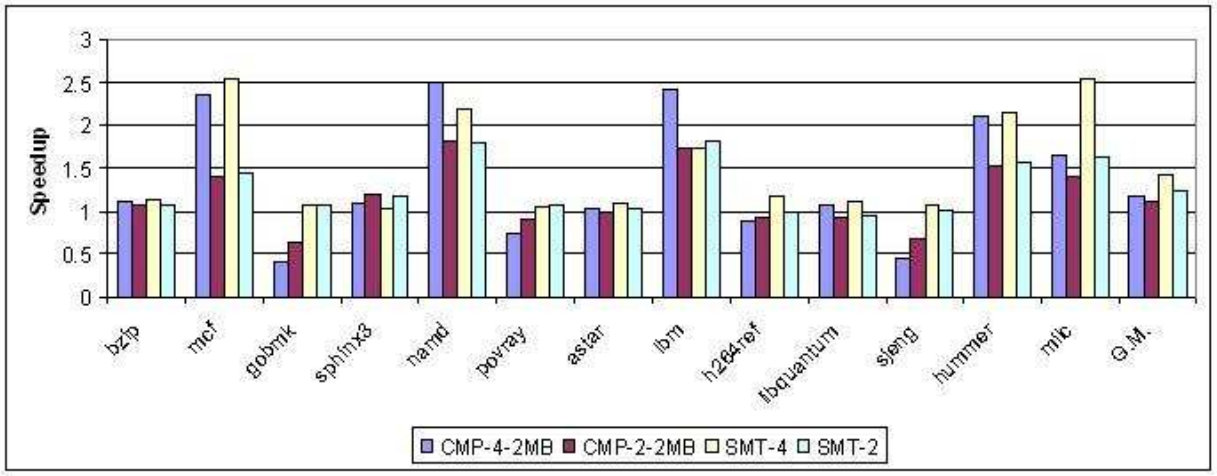


(a) ED^2 for Class 'A' benchmarks.

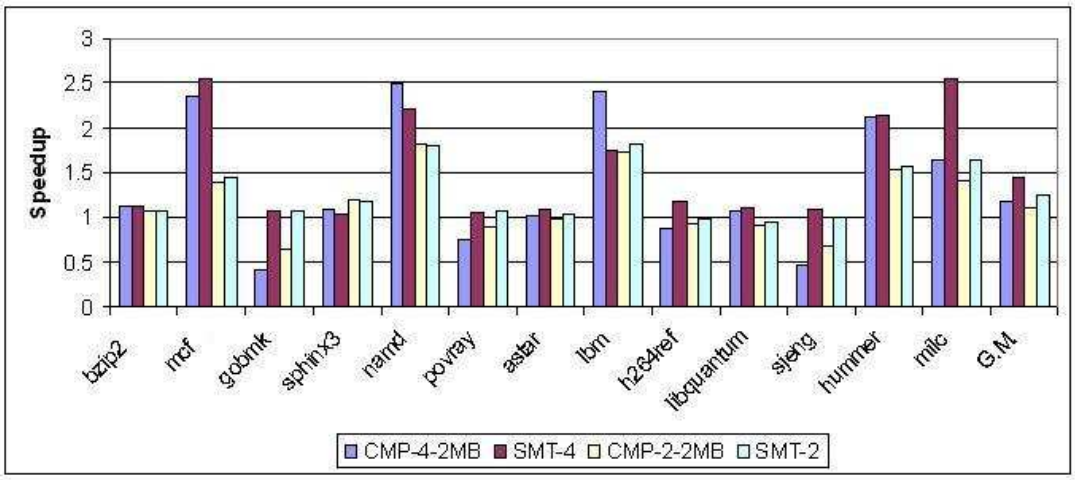


(b) ED^2 for Class 'B' benchmarks.

Figure 5.6: Comparison of ED^2 for Class 'A' and Class 'B' Benchmarks. ED^2 of CMP-4-2MB and SMT-4 configurations normalized to SEQ.



(a) Speedup for SPEC 2000 benchmarks.



(b) Speedup for SPEC 2006 benchmarks.

Figure 5.7: Impact of number of threads on speedup - comparison of speedup for CMP-4-2MB, CMP-2-2MB, SMT-4 and SMT-2 configurations.

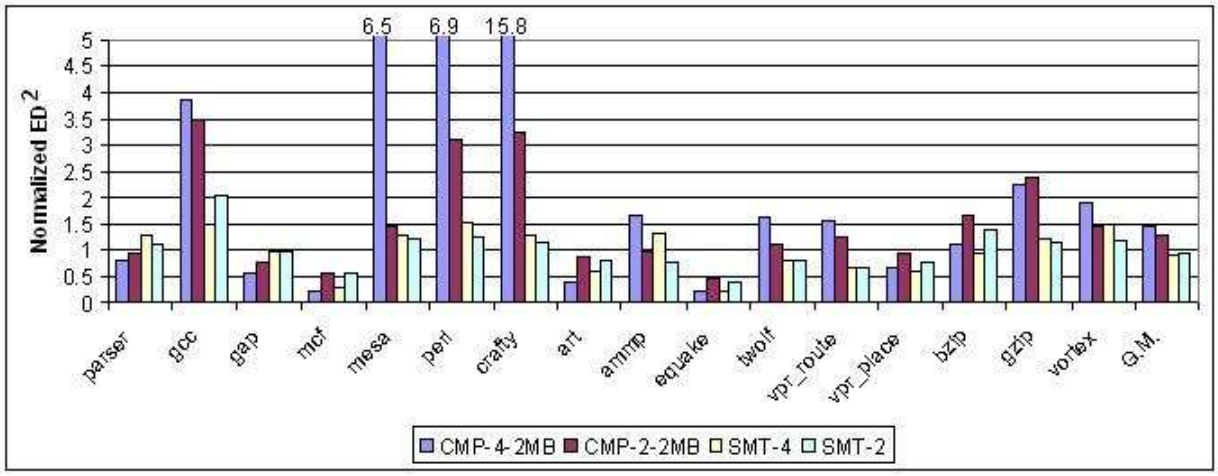
crafty, vortex, etc. When we consider the SPEC 2006 benchmarks which have better parallelism than SPEC 2000, the CMP-2-2MB slows down about 7% when compare to CMP-4-2MB. Due to this slowdown and its higher power overhead it suffers about 12% worser ED^2 than CMP-4-2MB. Overall the benchmarks the CMP-4-2MB is about 1% better than CMP-2-2MB.

The SMT-2 configuration can only support two threads, but suffers from fewer dependence violations. Due to its limited ability to exploit parallelism, it suffers from 9% slowdown when compared to SMT-4 for SPEC 2000 and about 19% slowdown for SPEC 2006 benchmarks. Due to its lower dynamic power overhead the decrease in efficiency (ED^2) is lesser when compared to its slowdown. It suffers from about 6% worser ED^2 for SPEC 2000 benchmarks and about 8% worser ED^2 for SPEC 2006 benchmarks. Across all the benchmarks the SMT-4 has about 7% better ED^2 than SMT-2.

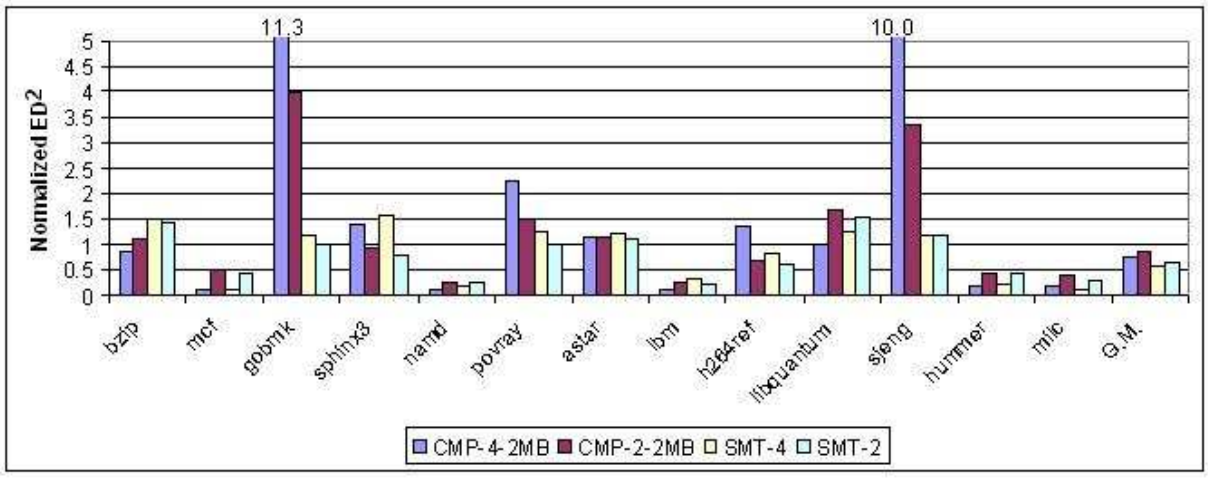
Impact of L2 size:

Reducing the number of cores in CMP allows us to use larger cores which lead to improved sequential region performance, but we lose performance in parallel regions. Another possible design choice is to reduce the L2 size, allowing the extra space to be used for larger cores. Fig. 5.9 compares the speedup of CMP-4-2MB with that of CMP-4-1MB which uses larger cores but smaller L2 size and in Fig. 5.10 comparison of ED^2 is presented.

In SPEC 2000 benchmarks, the CMP-4-1MB gains about 14% better speedup when compared to CMP-4-2MB. But due to its higher core complexity, it suffers from increased power overhead leading to 18% worser ED^2 when compared to CMP-4-2MB. Though the CMP-4-1MB showed better ED^2 for benchmarks like *crafty, mesa, perlbmk* which had large slowdown

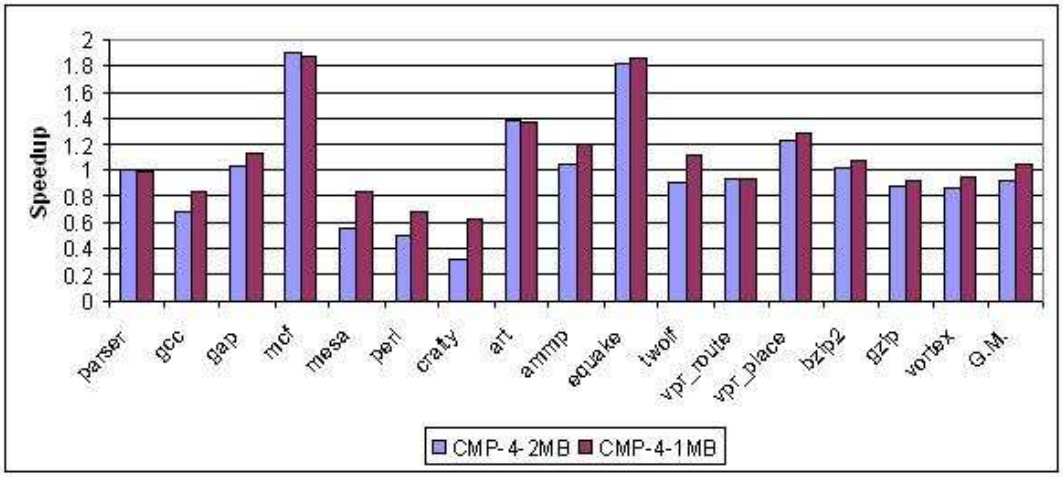


(a) ED^2 for SPEC 2000 benchmarks.

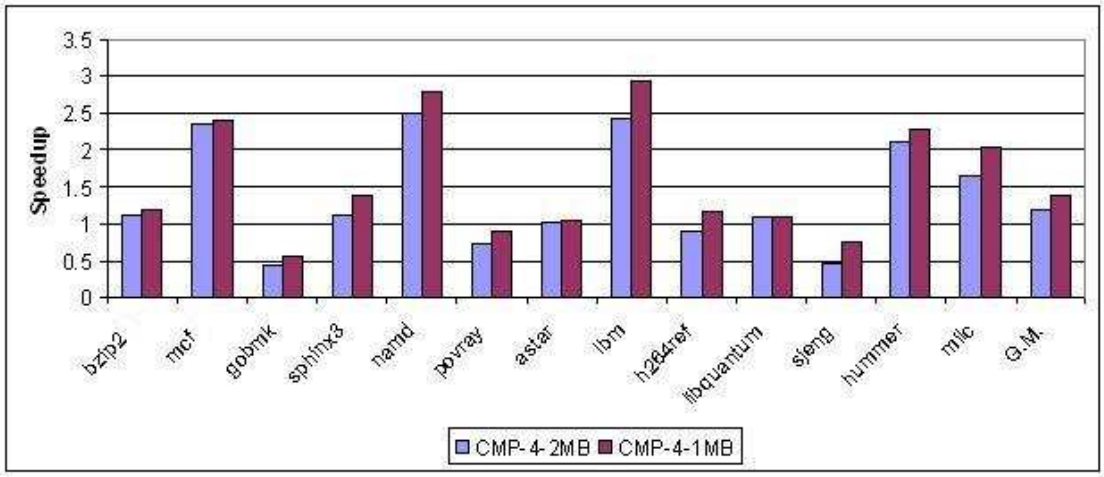


(b) ED^2 for SPEC 2006 benchmarks.

Figure 5.8: Impact of number of threads on ED^2 - comparison of ED^2 for CMP-4-2MB, CMP-2-2MB, SMT-4 and SMT-2 configurations.

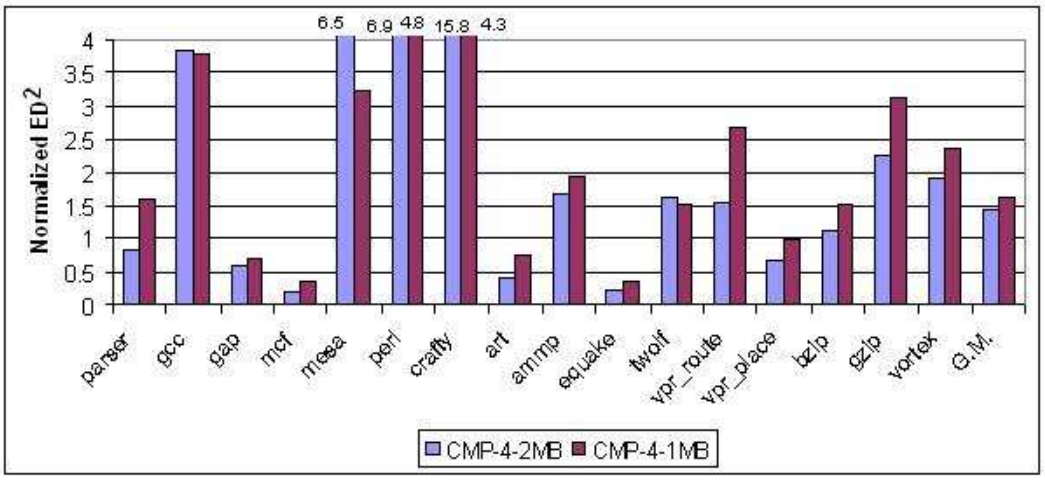


(a) Speedup for SPEC 2000 benchmarks.

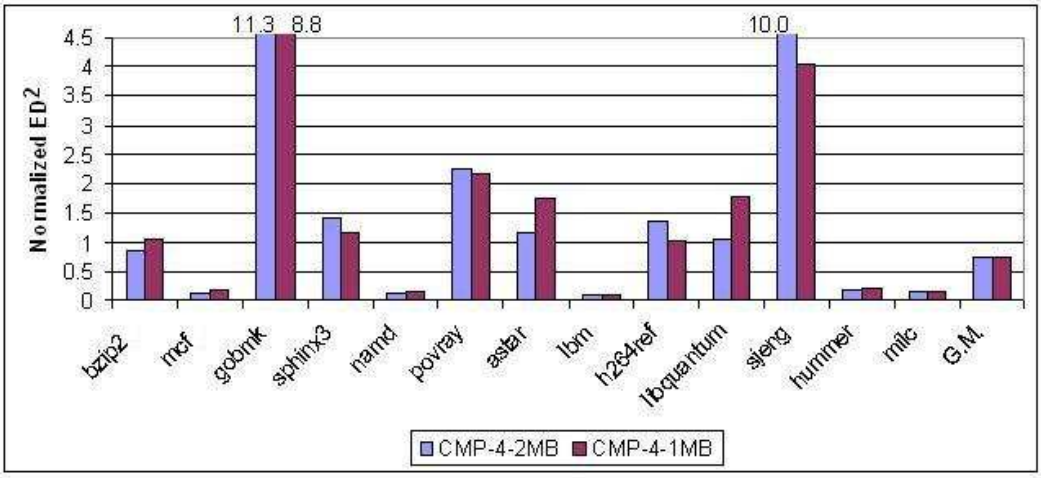


(b) Speedup for SPEC 2006 benchmarks.

Figure 5.9: Impact of L2 cache size on speedup - comparison of speedup for CMP-4-2MB and CMP-4-1MB configurations.



(a) ED^2 for SPEC 2000 benchmarks.



(b) ED^2 for SPEC 2006 benchmarks.

Figure 5.10: Impact of L2 cache size on ED^2 - comparison of ED^2 for CMP-4-2MB and CMP-4-1MB configurations.

for CMP-4-2MB, it suffers in many other benchmarks due to its higher complexity. Among SPEC 2006 benchmarks the CMP-4-1MB performs 21% better than CMP-4-2MB, but in terms of ED^2 it is about 2% worse than CMP-4-2MB. Over all the benchmarks, the CMP-4-2MB has 8% better ED^2 than CMP-4-1MB.

Impact of frequency: In our study, we had assumed the same clock frequency for all configurations. A simpler CMP core can be run at a higher frequency than in SEQ and SMT configurations. Though increasing frequency can lead to better performance, it leads to large increase in power consumption leading to worse ED^2.

Among the alternative design choices considered we found that SMT-4 still is the best possible configuration in terms of ED^2 for supporting TLS. Though CMP-2-2MB and CMP-4-1MB showed good speedup, they suffer worser ED^2 when compared to CMP-4-2MB.

5.5 Thermal behavior

The superscalar and the SMT-TLS architectures use complex cores with a large number of function units and large instruction window to exploit instruction-level parallelism or support the additional threads. These cores not only consume more energy, they can also generate thermal hotspots. On the other hand, the CMP-TLS architecture has distributed cores, and thus can potentially have smaller and less severe thermal hotspots. In this section, we analyze the thermal characteristics of three processor configurations—SEQ, SMT-4 and CMP-4-2MB.

The average and hotspot temperatures for each architecture are shown in Table 5.7. We have observed that the CMP-4-2MB configuration has the lowest average and hotspot temperatures,

while the SMT-4 has the highest average and hotspot temperatures Also the SMT-4 has more severe thermal spot temperature even compared to SEQ. In terms of hotspot temperature, the CMP-4-2MB configuration is about 2.5 degrees lower than that of the SMT-4 configuration; while SMT-4 configuration is about 0.8 degrees higher than that of the SEQ configuration.

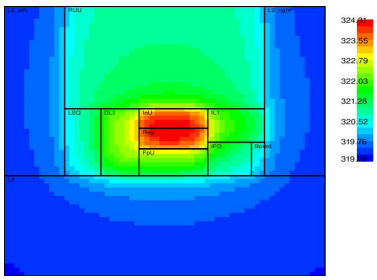
By observing the steady state temperature map for the SMT-4 and CMP-4-2MB configurations running `h264ref`, which has the one of the highest IPC among all benchmarks, we found that the main source of heat in all three configurations is the register file. The temperature maps are shown in Figure 5.11. The activity level in the register file of each CMP core is lower than the activity level of the central register file in SMT-4, thus leading to lower hotspot temperature. While both SMT-4 and SEQ have a centralized register file, the activity level in SMT-4 is higher due to the execution of speculative threads, thus it leads to a higher temperature.

5.6 Conclusions

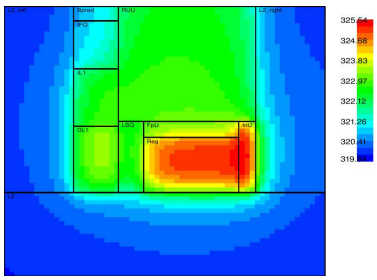
In this chapter, we compared the performance, power consumption, energy-delay-product and thermal effects of three architectures: superscalar, SMT and CMP, while holding the die area constant. We have identified major issues in each of the architectures and found that the SMT-TLS is more suitable for TLS applications. From our results, we have shown that:

- SMT-TLS can dynamically adjust its resources to achieve good TLS performance while not suffering significant slowdown in sequential code regions. The SMT-4 configuration achieves about 30% speedup over SEQ configuration.

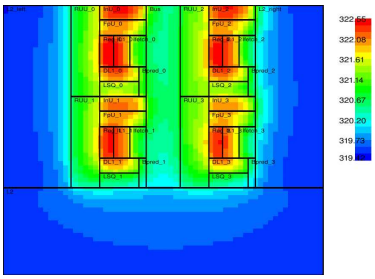
- Nevertheless, the good performance of SMT-TLS comes at the cost of about 25% increase



(a) Thermal map for the SEQ configuration.



(b) SMT configuration.



(c) CMP-4-2MB configuration.

Figure 5.11: Thermal map for various configuration (running h264ref).

in power consumption when compared to superscalar. But if we consider ED^2, the SMT-TLS outperforms both Supersclar and CMP-TLS architectures.

- The CMP-TLS architecture suffers due to poor sequential region performance. This can be improved by increasing the core complexity, but this increases power consumption. The CMP-4-2MB is the best CMP-TLS configuration which performs 34% worse than SMT-4 in terms of ED^2.

- The main disadvantage of SMT-TLS is that it creates more thermal stress than CMP-TLS due to its centralized register file.

Table 5.7: Thermal effects of TLS on three different architectures: SEQ, SMT-4 and the CMP-4-2MB in degree Celsius.

benchmark	SEQ		SMT-4		CMP-4-2MB	
	hotspot	average	hotspot	average	hotspot	average
SPEC 2000 benchmarks						
vpr_place	49.41	47.12	51.08	48.03	48.74	47.81
vpr_route	49.23	46.92	50.99	47.86	48.49	47.55
gcc	50.16	47.45	51.26	48.25	49.34	47.44
twolf	49.37	47.1	51.02	47.96	48.56	47.62
equake	49.23	46.89	50.93	47.87	48.49	47.54
ammp	49.26	46.91	51.05	47.88	48.7	47.58
gzip	49.93	47.64	51.38	48.57	49.34	47.44
bzip2	50.7	47.59	53.26	48.62	51.59	47.71
mcf	49.17	46.88	50.85	47.64	48.5	47.44
vortex	49.52	47.21	50.99	47.96	48.11	47.03
parser	49.42	47	50.93	47.81	48.81	46.78
perlbmk	49.91	47.51	51.11	48.04	47.86	46.8
crafty	50.17	47.86	51.22	48.33	46.83	46.02
art	49.09	46.81	50.89	47.7	48.28	47.27
mesa	49.93	47.29	51.13	48.05	48.99	47.45
gap	51.86	47.79	52.04	48.32	48.7	47.58
SPEC 2006 benchmarks						
bzip2	50.55	47.56	54.18	48.75	50.8	47.08
mcf	49.05	46.74	50.91	47.73	48.42	47.4
gobmk	49.75	47.35	51.12	48.1	48.3	47.15
namd	49.3	46.99	51.39	48.55	49.82	48.71
povray	49.65	47.32	51.08	48.03	48.64	47.66
astar	49.55	47.21	51.17	48.19	48.94	47.89
lbm	49.29	46.92	51.09	47.92	49.02	48.04
h264ref	52.22	48.24	53.75	49.22	50.61	48.29
libquantum	49.1	46.76	50.87	47.63	48.31	47.34
sjeng	49.1	46.91	51.21	48.11	49.30	48.20
hummer	49.5	47.21	51.73	49.05	50.21	49.15
milc	49.23	46.85	51.05	47.93	48.88	46.04
Average	49.76	47.23	51.42	48.15	48.94	47.47

Chapter 6

Heterogeneous TLS

In the previous chapter, we studied the efficiency of SMT and CMP based TLS architectures. We found that the characteristics of the benchmarks determine whether SMT or CMP is more efficient in supporting TLS. For benchmarks with sufficient parallelism, CMP is more efficient, otherwise SMT is more efficient. Thus, if a processor is designed as *homogeneous* SMT or CMP, it cannot exploit the parallelism with optimal efficiency across all possible benchmarks.

As the number of transistors in a chip keep increasing, processor designers try to make efficient use of the available transistors by including multiple cores in the same chip. With ever increasing die area, designers have a choice on the types of cores that can be included in the chip. Though most of the current multi-core systems use a *homogeneous* design where the same core is replicated, there are a few systems with *heterogeneous* cores. For example in IBM Cell processor, specialized cores called *SPE* which specialize in SIMD computations are added in addition to the *PPE* which acts as a control processor that services requests from the *SPE*s [5].

Several studies have examined the design of such *heterogeneous* multi-cores [44, 45, 46] and how to allocate the application into such *heterogeneous* cores. In most papers, different cores with varying complexity and frequency are combined together to form a *heterogeneous* multi-core processor. In our case, in order to exploit TLS efficiently for all applications we need both SMT and CMP within a *heterogeneous* multi-core. In this chapter we show the potential for using such SMT-CMP based *heterogeneous* multi-core to efficiently exploit TLS.

In a typical *heterogeneous* multi-core, the performance of the different ”threads” are monitored and the runtime system would decide which core to be used to improve efficiency. In most papers, only multi-programmed workloads are considered where entire applications are mapped into available cores. In our case, our speculative threads are fine grained and may require low-overhead runtime techniques. Detailed design of such hardware techniques to exploit such *heterogeneous* multi-core is beyond the scope of this thesis.

The rest of the chapter is as follows: In Section 6.1 we give a brief overview on the related work, in section 6.2 we show the potential improvement with the use of *heterogeneous* multi-core technique, in Section 6.3 we discuss the overheads of exploiting such *heterogeneous* multi-core technique and in Section 6.4 we present our conclusions.

6.1 Related work

Several studies have examined the problem of scheduling multiple tasks on a heterogeneous multi-core. Kumar *et*. al [46] used a dynamic scheduling approach, where the performance

of different tasks are studied during periodic profile phases. The performance information collected during the profile phase is used to map the different tasks to the available heterogeneous cores to optimize their efficiency. The drawback is that in the profile phase, different combinations of core allocation need to be sampled to determine ideal core selection. This process can incur excessive overhead.

John *e*t. al [44], collected different program characteristics like dependency distance, data reuse distance, etc during a profile run. Fuzzy logic is then used to calculate the suitability of the different tasks for the different heterogeneous cores. With this technique, there is no need for the expensive runtime sampling proposed in [46]. Crowley *e*t. al [45] also avoid the expensive profile phase and instead of trying different core combinations, they use the dynamic IPC (instructions per clock cycle) to decide the suitability for different tasks on each core.

In all the above studies multi-programmed workloads are considered, while in our thesis the focus is on TLS. Also in contrast to other previous work which typically use different cores of varying complexity or frequency, in this chapter we consider a SMT-CMP based heterogeneous multi-core architecture.

6.2 Potential for heterogeneous multi-core

In the previous chapter we studied the efficiency of TLS in SMT and CMP architectures. The results of our study is shown again in Fig. 6.1. As shown in Fig. 6.1, depending on the characteristics, each benchmark is either more efficient on a SMT or on a CMP. Fig. 6.1 shows that the geometric mean ED^2 of the best configurations for each benchmark is about 10% better

than the ED^2 of SMT configuration which gives the best overall efficiency. This indicates the potential efficiency that can be gained if each benchmark is able to select either SMT or CMP configuration based on an *Oracle* mechanism.

This shows that to extract TLS in the most efficient way, we need to support for both SMT and CMP. Such an heterogeneous multi-core is shown in Fig. 6.2. With this design, the benchmark can choose either SMT or CMP based on its characteristics to achieve optimal efficiency.

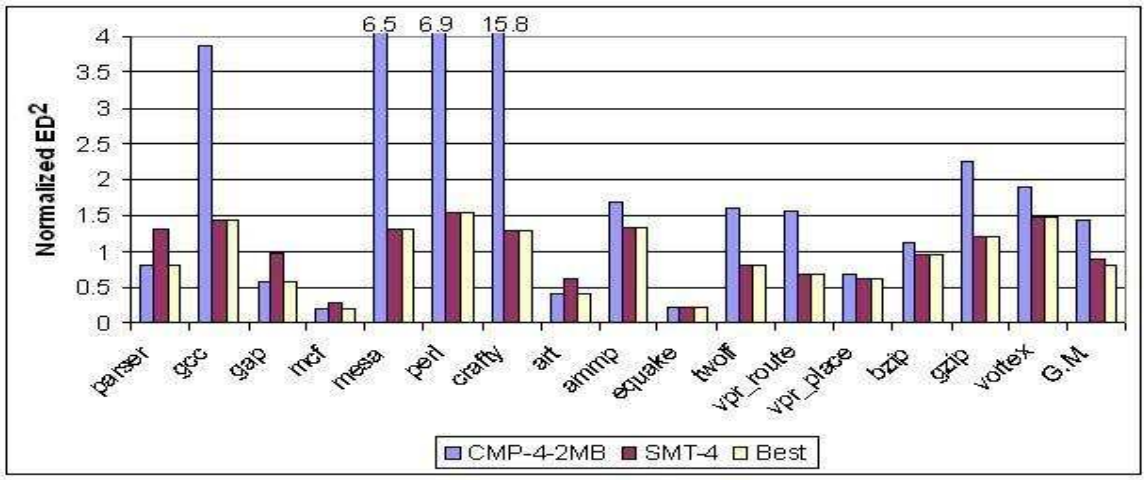
In Fig. 6.1, we showed the potential of mapping either applications to the best configuration. But within each benchmark different regions of code could perform better in different architectures. Thus we could further improve the efficiency if individual regions in each benchmark are able to choose the most suitable architecture.

To understand the potential of such fine grained selection of cores, we simulated each benchmark in three different configurations:

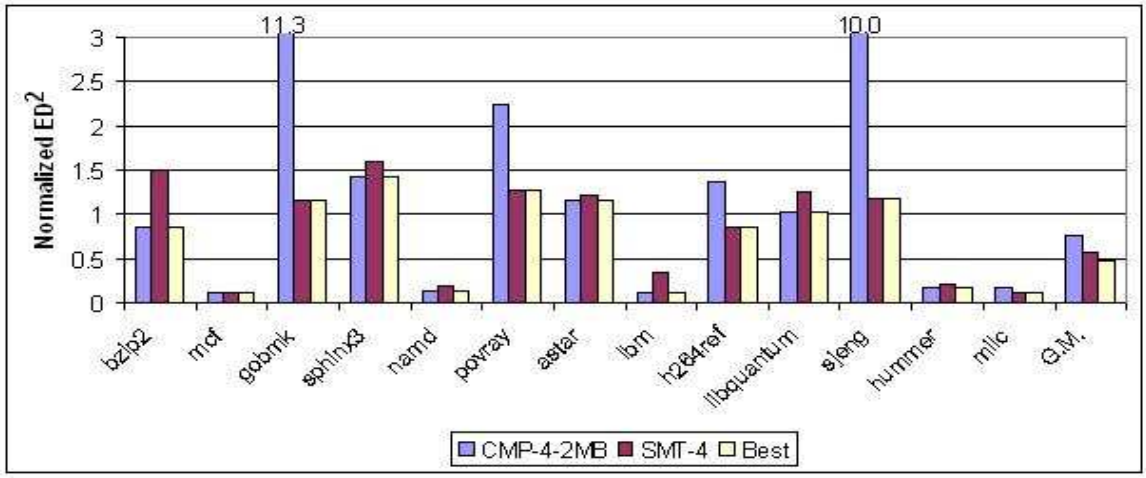
- SMT-TLS: Here the parallelized benchmark is run on the SMT core with TLS enabled.
- CMP-TLS: Here the parallelized benchmark is run on CMP core with TLS enabled.
- SMT-noTLS: Here the parallelized benchmark is run on the SMT core with TLS disabled. This configuration is useful when the CMP and SMT based TLS configurations suffer from frequent squashes and we need to disable TLS to avoid slowdown.

The three configurations are compared with the SEQ configuration used in chapter 5, running the non-parallelized benchmark.

Say a benchmark contains three regions - R_1,R_2 and R_3. Let SMT-TLS be the most efficient



(a) Normalized ED^2 for SPEC 2000 benchmarks.



(b) Normalized ED^2 for SPEC 2006 benchmarks.

Figure 6.1: Comparison of Normalized ED^2 of CMP, SMT configurations with the best configuration for each benchmark.

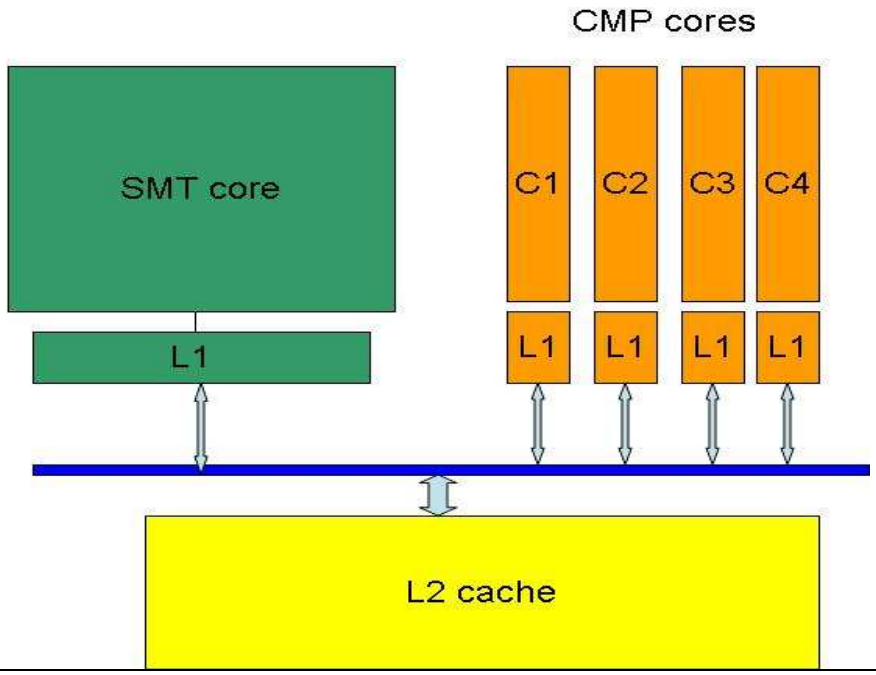


Figure 6.2: Heterogeneous multi-core architecture.

configuration for R_1, CMP be the best configuration for R_2 and SMT-noTLS be the best configuration for R_3. In an ideal heterogeneous multi-core architecture the benchmark would execute the different regions in their best configuration. To calculate the best efficiency obtained by such ideal switching between cores, the number of cycles and power consumed for each region in the benchmark which executing in their corresponding best configuration are added and overall ED^2 is calculated. For example let C_{m1} and P_{m1} be the time taken and the power consumed by region R_1 when running on its best configuration SMT-TLS. Similarly let C_{c2} and P_{c2} be the time taken and the power consumed by region R_2 and let C_{s3} and P_{s3} be the time taken and the power consumed by region R_3. Now the total cycles taken by choosing the ideal configuration for each region is $C_{m1} + C_{c2} + C_{s3}$ and the corresponding power consumption is $P_{m1} + P_{c2} + P_{s3}$.

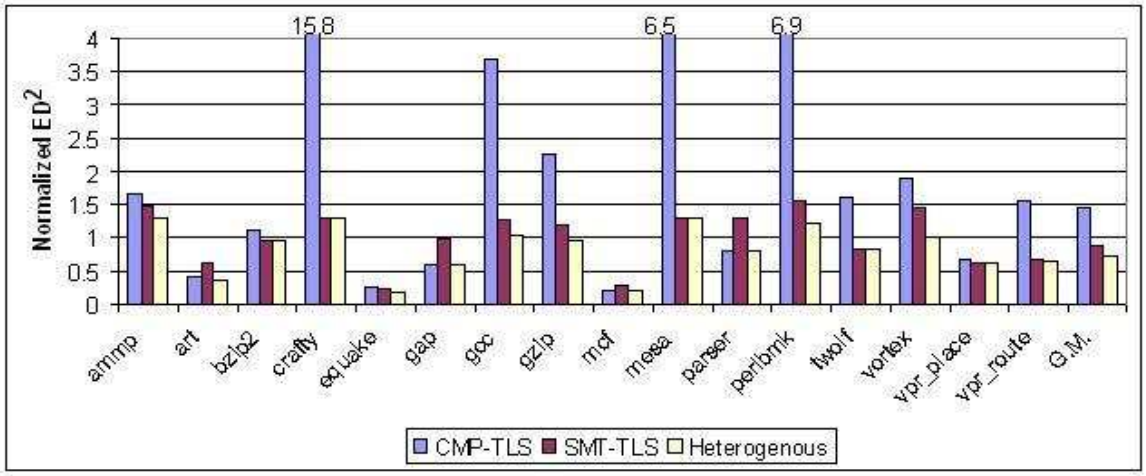
From this the ideal ED^2 for the benchmark calculated.

Fig. 6.3 compares the estimated ED^2 calculated for an ideal heterogeneous multi-core with the SMT-TLS and CMP-TLS configurations. Fig. 6.3 shows that the heterogeneous multi-core is about 16% more efficient than the SMT-TLS configuration which is the most efficient homogeneous configuration. When compared to results in Fig. 6.1, fine-grained switching between cores to optimize ED^2 results in about 6% additional speedup when compared to mapping the entire benchmark to a particular configuration.

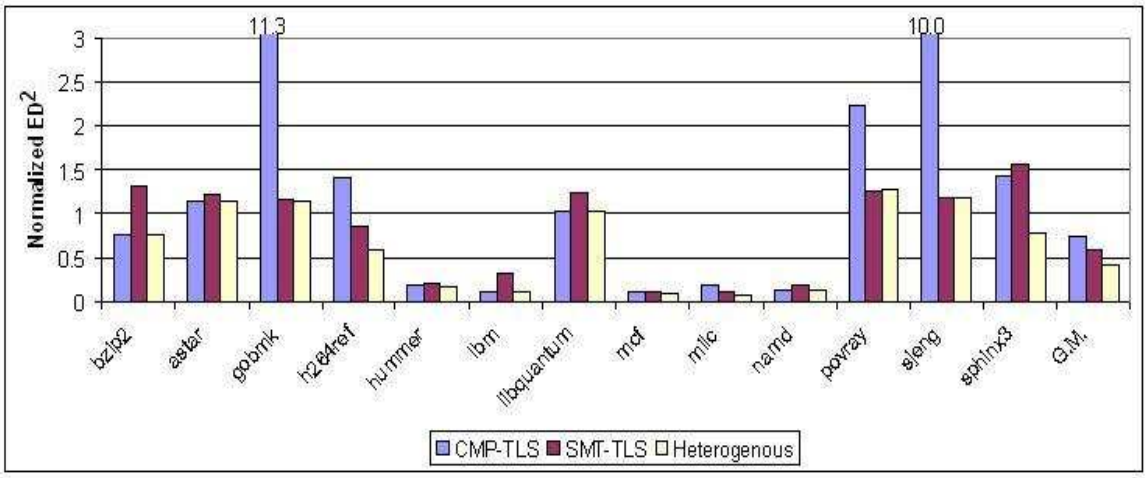
The efficiency obtained by using the heterogeneous multi-core can be classified into three-categories based on its source:

Efficiency from core selection: In benchmarks *art, bzip2, crafty, gap, mcf, mesa, parser, twolf, vpr_place, vpr_route, astar, gobmk, hummer, lbm, libquantum, namd, povray and sjeng* the efficiency of heterogeneous configuration is the same as the best homogeneous configuration (either SMT-TLS or CMP-TLS). For these benchmarks, based on their characteristics, a compiler or an advanced user can potentially assign the entire benchmark to the best suitable core. A simple hardware technique that can decide which configuration to use based on an initial profile phase can also be used.

Efficiency from speculative thread pruning: In benchmarks *art, gzip, perlbmk and sphinx3* some of the loops suffer from frequent squashes leading to poor efficiency in both SMT-TLS and CMP-TLS configurations. The efficiency gain due to heterogeneous multi-core for these benchmarks shown in Fig. 6.3 is mainly achieved by disabling TLS for such



(a) Normalized ED^2 for SPEC 2000 benchmarks.



(b) Normalized ED^2 for SPEC 2006 benchmarks.

Figure 6.3: Comparison of Normalized ED^2 of CMP, SMT configurations with the predicted ED^2 of heterogeneous multi-core.

loops. To obtain this benefit, we need a runtime monitoring system that can prevent inefficient TLS. But these benchmarks still need heterogeneous multi-core as they need to choose the best suitable configuration (Efficiency from core selection).

Efficiency from core switching: For benchmarks *ammp, equake, gcc, h264ref and milc*, different loops need to be assigned to different configurations to obtain the ideal efficiency. To exploit this efficiency, we need to develop low-overhead hardware techniques that can monitor the performance of each region of code and dynamically map them to different configurations based on their predicted efficiency on different configurations.

From the discussion above we can see that the efficiency from core switching is the hardest to extract as it requires complex runtime monitoring support. In the next section we study the overhead involved in extracting this efficiency.

6.3 Overhead in using heterogeneous multi-core

Fig. 6.4 shows the overhead involved in the execution of two regions Region R1 and R2 which require a switching between the two configurations. The different phases on execution when switching between cores are:

- Run phase in SMT: Region R1 is mapped to SMT configuration to maximize its efficiency. When executing in the SMT configuration the CMP configuration is in power-off phase.

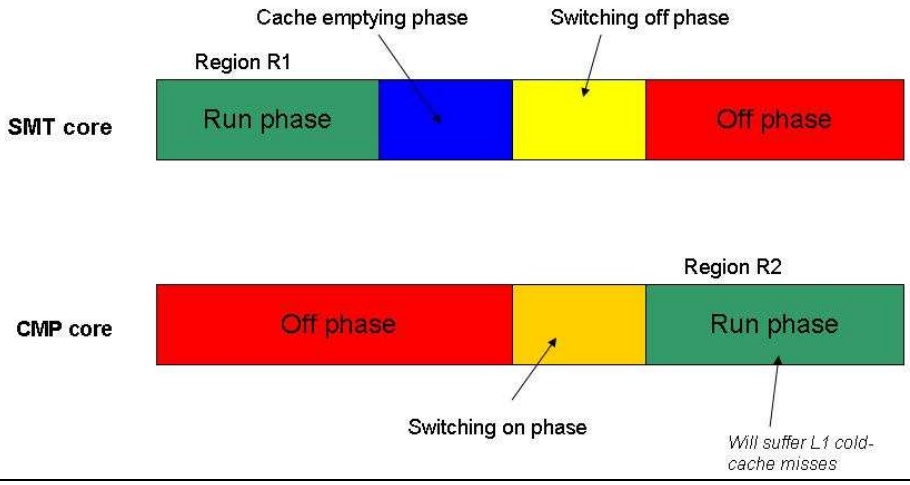


Figure 6.4: Different phases of execution when switching between cores.

- Cache emptying phase: After executing Region R1 in the SMT configuration, we need to switch to the CMP configuration to obtain the best efficiency for Region R2. When switching from the SMT configuration to the CMP configuration, we need to power-off (or put in a low power state) the un-used SMT configuration to reduce power wastage in the un-used core. But the L1 data cache is *write-back* (as discussed in chapter 4) and can contain *dirty* data that need to be written to the L2 cache.

- Switching-off phase: After the *cache-emptying phase*, the SMT core can now be safely switched off. Switching off a core (or putting in a low power state) may require several cycles.

- Switching-on phase: To use the CMP cores, they need to be powered-on. Similar to the Switching off phase, switching on the cores require several cycles. The Switching-on phase of CMP can be overlapped with the Switching-off phase of SMT as shown in Fig. 6.4.

- Run phase in CMP: Now the Region R2 starts executing in the CMP configuration. Since the caches in the CMP cores are initially empty, the CMP configuration would suffer from *cold* cache misses when executing Region R2.

From the above discussion, it is clear that switching between cores involves several overheads. To understand the impact of the switching overhead, we first present the results when assuming *ideal* switching.

6.3.1 *ideal* switching:

Here we assume that the the L1 cache can be emptied and the cores can be switched on or off instantaneously. So the different overheads shown in Fig. 6.4 do not occur in *ideal* switching. Nevertheless when the Region R2 starts executing in the CMP configuration it would still suffer

from *cold* cache misses as shown in Fig. 6.4.

Input: List of regions R_i in the order of execution, Efficiency ED^2 for each region under SMT-TLS, CMP-TLS and SMT-noTLS configurations – E_{mi}, E_{ci} and E_{si} respectively. Lower bound of the region size

Output: Ideal allocation for each region

forall *Region* R_i **do**

  /*Find the best allocation.*/ Best Efficiency for R_i = min(E_{mi},E_{ci},E_{si});

  Best allocation for R_i = configuration corresponding to the Best Efficiency;

end

SizeNewRegion[0] = size of R_0;

foreach *Region* R_i **do**

  if *CurrentNewRegion->size* $\leq$ *LowerBound* **then**

    /*Include current Region R_i into the CurrentNewRegion*/

    CurrentNewRegion->size += size of R_i;

    CurrentNewRegion $\sum$= R_i;

  end

  else

    /*CurrentNewRegion is large enough. It could contain multiple R_is*/ **foreach** *Region* R_j, $R_j \in$ *CurrentNewRegion* **do**

      totalSeq += size of R_j if Best allocation for R_i is SMT-noTLS;

      totalSmt += size of R_j if Best allocation for R_i is SMT-TLS;

      totalCmp += size of R_j if Best allocation for R_i is CMP-TLS;

    end

    CurrentNewRegion->bestAllocation = configuration of max(totalSeq,totalSmt,totalCmp);

    CurrentNewRegion = create new region;

  end

end

return the list of new regions created ;

Algorithm 1: *FormLargerRegions* to get combine regions with smaller size into larger regions.

Consider 3 regions R_1,R_2 and R_3. Lets assume that R_1 and R_3 are mapped to SMT configuration and let R_2 be mapped to CMP configuration. In our heterogeneous multi-core mechanism, we need two switches – SMT to CMP to execute R_2 and then from CMP back to SMT to execute R_3. If the size of R_2 is small, the efficiency gain by using CMP to run R_2 may not be high enough to offset the overhead due to *cold* cache misses that occur after each switch. In this case it may be beneficial to not switch to CMP and let R_2 run in SMT configuration. To reduce such un-beneficial switching, we use the *FormLargerRegions* algorithm shown in Algorithm 1.

The algorithm basically combines the smaller regions into a larger region and allocate the best possible configuration based on the efficiency information of the smaller sub-regions. The list of larger regions formed is feedback to the simulator. The simulator allocates the different code regions to specific configurations as determined by the *FormLargerRegions* algorithm.

In Fig. 6.5, the efficiency of different benchmarks when assuming *ideal* switching under different lower bounds are compared with the efficiency of best homogeneous configuration in each benchmark. Also we compare the efficiency of *ideal* switching with predicted efficiency calculated in Section 6.2.

The impact of fine-grained core switching is shown in bars corresponding to 100-cycle regions. With the region size lower bound of 100 cycles, the switching is frequent, leading to *cold* cache misses. For benchmark *ammp*, the impact is less due to its large region size. But in other benchmarks the fine-grained switching leads to poor efficiency. In *gcc and h264ref* which have many regions with small size, the overhead leads to an ED^2 which is worse than that of the

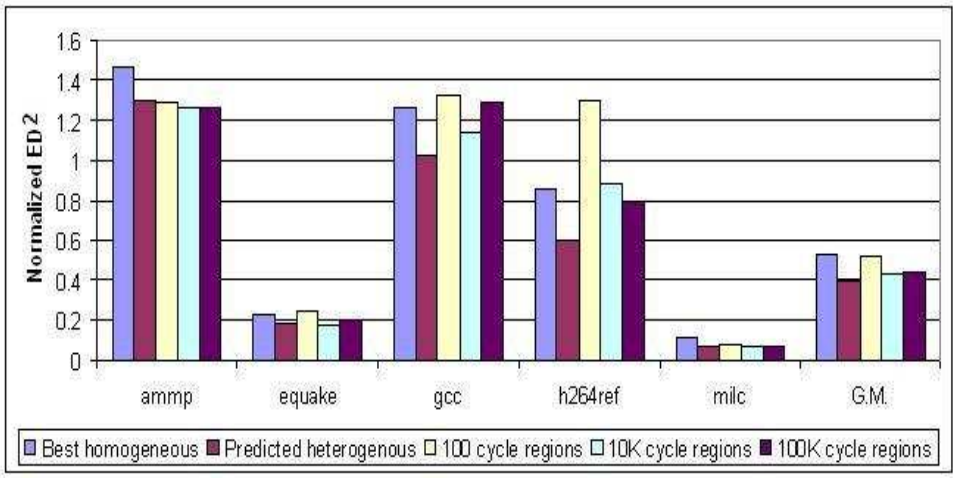


Figure 6.5: Comparison of *ideal* switching based heterogeneous multi-core architecture with best homogeneous configuration.

best homogeneous configuration. Overall the 100-cycle region *ideal* switching does not gain any efficiency and it approximately has the same efficiency as that of the best homogeneous configuration.

The impact due to fine-grained switching is reduced when we use the 10,000 cycle lower bound for region size. It performs about 10% better than the best homogeneous configuration and comes within 4% of the predicted efficiency calculated in the previous section.

When we further increase the lower bound for region size to 100,000 cycles, the efficiency becomes worse. When allocating at a larger granularity, many of the smaller sub-regions within the larger region are not allocated in the most optimal way. With increased region size of 100K cycles, significant number of smaller sub-regions are allocated to sub-optimal configurations leading to worser efficiency than the 10K case by about 1%. This effect is significant in *gcc* while in *h264ref* the larger region size has a small positive effect due to reduced *cold* cache

misses. Overall the lower bound region size of 10K cycles is the most optimal size leading to best efficiency.

6.3.2 Impact of switching overhead:

In the above study, we assumed *ideal* switching and studied the impact of *cold* cache misses. Here we include all the overhead involved in switching as shown in Fig. 6.4. We assume that the number of cycles to write back the *dirty* lines in the L1 data cache is equal to the number of *dirty* lines in the cache. The number of cycles to switch on/off a configuration is assumed to be 1000 cycles. Similar to the *ideal* switching study, we feedback the best allocation for each region to the simulator, which assigns the different code regions to different configurations based on the feedback information. For each benchmark, we use the region size that lead to the best efficiency with *ideal* switching.

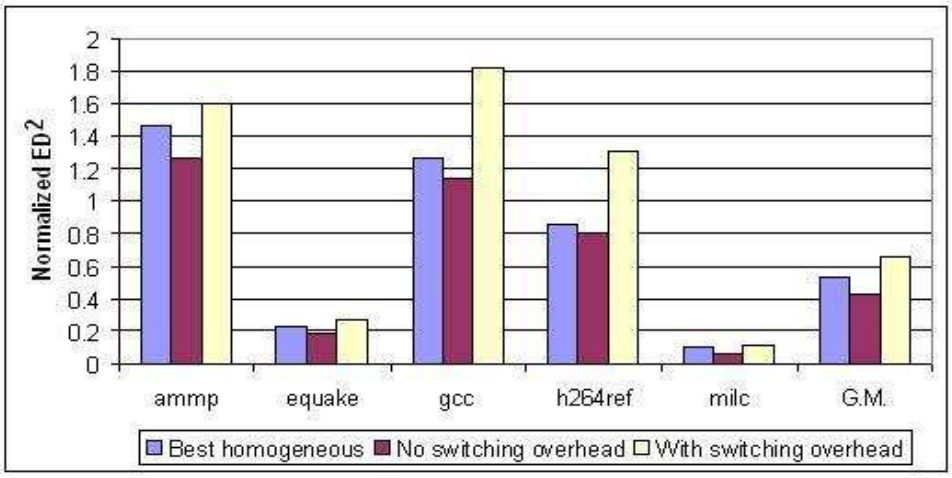


Figure 6.6: Impact of switching overhead.

Fig. 6.6 compares the efficiency of the heterogeneous architecture when all overheads are

included with the efficiency under *ideal* switching. Also the comparison with the best homogeneous configuration is shown. From Fig. 6.6 it is clear that the overhead due to core switching is significant and leads to about 24% worser efficiency when compare to the efficiency with *ideal* switching. Even the best homogeneous configuration out-performs the heterogeneous architecture by 13%.

6.3.3 Reducing switching overhead:

From the discussion above, it is clear that the overhead due to thread switching is significant and need to be reduced. One possible way of reducing overhead is to overlap some of the phases involved in switching. The hardware can start the write-back process and the switching-on phase ahead in time by predicting the imminent end of the current region. Also compiler generated warm-up instructions could be used to warm-up the cache before switching. Design of such techniques targeted at reducing the switching overhead is beyond the scope of this thesis.

6.4 Conclusions

In this chapter we presented the SMT-CMP based heterogeneous architecture. We showed the potential improvement in efficiency that can be achieved by the use of such heterogeneous architecture. We showed an overall potential of about 16% (in terms of ED^2) when compared to the SMT configuration and 51% potential when compared to the CMP configuration.

We classified the efficiency improvement obtained according to the source of the improvement. For the increase in efficiency that requires fine-grained switching between cores, we

studied the impact of switching. Using an *ideal* switching strategy we showed a potential of about 10% when compared to the best homogeneous configuration.

But, when we include all the switching overheads, the efficiency drops to a negative 13% when compared to the homogeneous configuration. This shows the need to develop techniques that can reduce the switching overhead.

Chapter 7

Increasing scalability with multi-level speculative threads

In previous chapters we explored the efficiency of TLS architectures. As we saw in chapter 5, the main factor that leads to inefficiency in TLS is the lack of parallelism in the loops selected. When parallelism is limited, not all cores are utilized leading to poor efficiency. With increasing number of cores in future multi-core processors it is imperative to develop techniques that can utilize all the available cores. Most TLS techniques focus on extracting speculative threads from single loop level or a single function continuation. But to utilize all the cores available we need to extract parallelism available at all levels of loops and functions. Consider a high-coverage loop in benchmark *povray*,

```
/*csg.cpp*/
330:  static int All_CSG_Merge_Intersections (OBJECT *Object, RAY *Ray,
```

```
ISTACK *Depth_Stack)
...
350:  for (Sib1 = ((CSG *)Object)->Children; Sib1 != NULL;
Sib1 = Sib1->Sibling) { //Loop-1
...
364:     for (Sib2 = ((CSG *)Object)->Children; Sib2 != NULL &&
inside_flag == true; Sib2 = Sib2->Sibling){ //Loop-2
```

Here, loop-1 has an average iteration count of approximately 4 while the average iteration count of loop-2 is 2. Both these loops are possible candidates for parallelization, but if we parallelize only at one loop level all cores in a 8-core CMP cannot be utilized. Apart from low iteration count, loops could also suffer from large synchronization delay, large number of mis-speculations, etc which lead to poor utilization of available cores. So to maximize performance we need to extract TLS at multiple levels.

Extending TLS to multiple levels introduces several architecture and compiler challenges. Compared to single-level TLS, architecture and compiler techniques to support multi-level TLS have not been well understood. Supporting TLS at multiple levels often requires complex hardware support [47]. This chapter addresses this issue by proposing novel compiler and architecture techniques that can efficiently extract parallelism at multiple levels with minimal hardware cost. To support multi-level TLS, any TLS system should address two key challenges: 1. maintaining the sequential order and 2. efficient allocation/scheduling of threads to available cores.

To ensure correctness of the application, the TLS threads need to be committed in the sequential program order. In single-level TLS, the logic to commit the speculative threads in-order is relatively simple, as the threads are committed in the same order as they were forked. In multi-level TLS, the outer loop iterations are forked earlier than the inner loop iterations. But to maintain sequential commit order, the inner loop iterations are committed earlier. Renau *et. al* [47] proposed a time-stamp based architecture to support such out-of-order threads. The drawback of this approach is that every TLS thread (for the loop-based TLS used in this thesis, every iteration of an inner loop) creates a new version of data and all the speculative versions need to be maintained till the thread commits. Since the L1 data cache's associativity is used to maintain the different versions, too many versions cannot be maintained leading to stalling/squashing of speculative threads. In contrast in this chapter we propose a novel SpecMerge architecture where, when an inner level thread completes, its state is merged with the state of the outer level thread similar to the "closed" nested Transactional Memory (TM) [79]. This reduces the number of versions that need to be maintained, thus reducing the stalling/squashing effect and it also reduces hardware complexity.

If the TLS threads from multiple levels are not efficiently scheduled it is possible that a low-performing loop/function could consume all the available cores leading to overall degradation of performance. Scheduling and loop allocation for nested loops have been well studied in the context of conventional parallel loops. We use an approach similar to the OPTAL algorithm used by Polychronopoulos *et. al* [80] to parallelize arbitrary DOACROSS loops. The SpecOPTAL algorithm proposed in our chapter uses the predicted performance of each level to schedule

speculative threads from nested regions to maximize the performance of the entire application. Renau *et. al* [47] used Dynamic Task Merging, an ad-hoc technique where threads which causes too many squashes are disabled at runtime. But in addition to the rate of mis-speculation the TLS performance is affected by other factors such as synchronization delay, iteration count of loops and cache behavior. The SpecOPTAL algorithm proposed considers all these factors at compile time and appropriately allocates the available cores to maximize performance. Since the allocation is done statically, our technique greatly reduces the hardware complexity.

Using our detailed out-of-order simulator and Open64 based compiler framework, we show the effectiveness of our compiler and architecture techniques to exploit TLS at multiple levels. Though in this chapter we apply our techniques to nested loops, the same technique can be applied to extract other types of multi-level TLS.

The rest of the chapter is organized as follows: In Section 7.1 we present a description of related work. In section 7.2 we motivate the need for multi-level TLS by showing the limitations of single-level TLS. Section 7.3 describes our compiler based loop allocation framework and Section 7.4 describes our SpecMerge architecture. In Section 7.5 we present our evaluation results and in Section 7.6 we conclude the chapter.

7.1 Related work

Compared to single-level TLS, the architecture/compiler design for supporting TLS at multiple loop levels (or function levels) have not been well understood. Several papers using ideal machine models [16, 61] have shown the potential of different types of TLS. Oplinger *et. al* [16]

show that by exploiting TLS from nested loops, the harmonic mean performance can increase from 1.6 to 2.6.

Renau *et. al* [47] proposed architecture modifications to support out-of-order TLS threads. In [47], all potentially parallel loops are parallelized and the architecture design relies on run-time hardware tracking mechanisms to dynamically allocate resources. Using hardware tables, the rate of mis-speculation for each loop is tracked and if a loop squashes often, the threads from the loop are merged, thus freeing the resources for other potential loops. Such runtime tracking mechanisms require complex hardware support. Also we saw that the performance of the loop depends on various other factors such as synchronization delay, iteration count and cache performance.

In this thesis, the compiler statically allocates resources by considering all the loop characteristics. Our approach of static allocation of resources is similar to the approach used in scheduling nested loops in traditional DOACROSS/DOALL parallelism [80]. The main difference is that, in our approach, we have to consider the impact of speculation failure in addition to factors like initiation interval (synchronization delay) and iteration count.

In our SpecMerge architecture the inner thread merges its state with outer-thread on completion to avoid maintaining too many versions in the cache. This approach of ”merging” of state with outer loop is similar to ”closed-nest” Transaction memory (TM). In [79], the cache tags are augmented with Read and Write bits for each loop nesting level to track the dependences and a stack of log frames hold the undo logs for each nesting level. Instead of using undo logs, we use the level-1 data cache to maintain the speculative values similar to other cache based

TLS approaches [49]. Also we use fewer bits per cache line to track dependences.

7.2 Limitations of single-level TLS

Lets say the program contains an arbitrary loop nest (L_1,L_2, ... L_i ... L_N) where L_N is the inner most loop. Let the coverage of the loops be (C_1,C_2, ... C_i ... C_N) and let the estimated speedup be (S_{1p},S_{2p}, ... S_{ip} ... S_{Np}) using p cores. For single-level TLS, the compiler loop selection algorithm should select a loop L_i such that it satisfies:

$$(1 - C_i + \frac{C_i}{S_i}) \leq (1 - C_j + \frac{C_j}{S_j}) \forall j = 1 to N \tag{7.1}$$

When we introduce multi-level TLS, say for example 2-level TLS, the objective is to find two loops L_m and L_n (say L_m is the outer loop, $m < n$) with allocation p_m and p_n such that:

$$(1 - C_m) + \frac{C_m - C_n + (\frac{C_n}{S_{np_n}})}{(S_{mp_m})} \leq (1 - C_i + \frac{C_i}{S_i}) \tag{7.2}$$

Following scenarios could occur:

1. Condition in (2) is never satisfied. This indicates either that the performance of L_i is hard to beat or there are no good loops other than L_i in the loop nest.

2. L_m is outer loop of L_i. ($m < i$). Since L_m was not selected for single-level TLS (1), it shows that $S_{mp} < S_{ip}$, but $S_{mp_m} > S_{ip_m}$ as L_m was selected for 2-level TLS. This indicates that the performance of L_m does not scale with increase in processors.

3. L_m is inner loop of L_i. ($m >= i$). This shows that the two inner loops have good performance that in combination are able to beat the performance of higher coverage L_i. This implies poor scalability of loop L_i.

Thus the important factor that requires support of multi-level TLS (in scenario 2 and 3) is the presence of loops that have poor TLS performance. Though these loops have some speedup, the performance is not high enough to use all the available cores efficiently.

7.2.1 Scalability in SPEC 2006:

In Chapter 3, we studied the performance of SPEC 2006 benchmarks using four cores. In Fig. 7.1 we vary the number of cores and show the effect of increasing the cores for these SPEC 2006 benchmarks.

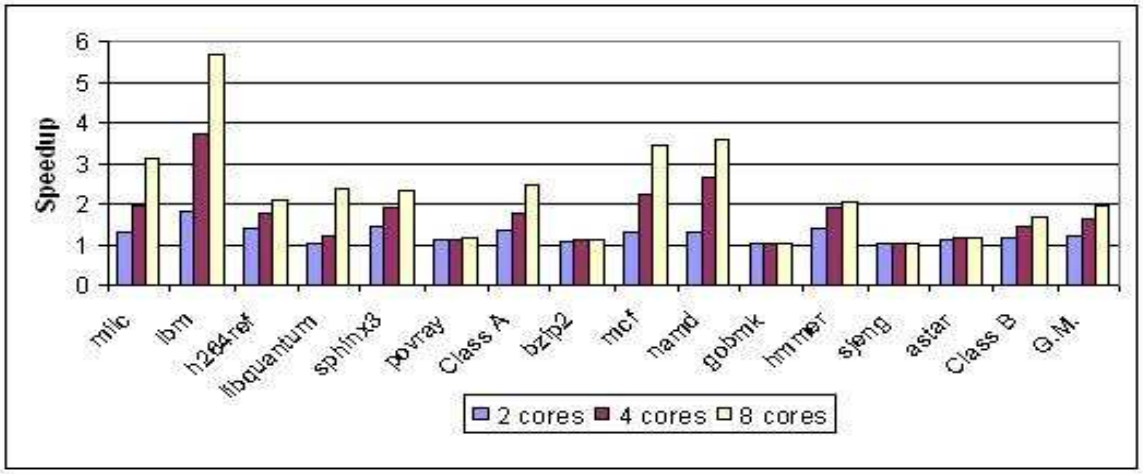


Figure 7.1: Effect of increasing number of cores

When the number of cores is increased from two to four, the geometric mean of the speedup increases by about 35%; when increased further to eight cores, the performance increase is 33%.

Among class **A** benchmarks, LBM,SPHINX3, H264REF and LIBQUANTUM contain important loops that have large iteration count and substantial amount of parallelism, thus the performance of these benchmarks is able to scale with the number of cores. In LIBQUANTUM, the super-linear performance gain is due to cache prefetching effect between the speculative threads.

Among class **B** benchmarks, NAMD shows good scalability and MCF benefits from cache prefetching effect as the number of threads increases. Unfortunately, none of the other benchmarks are scalable: HMMER suffers from frequent synchronization; POVRAY and BZIP2 suffers from small trip counts; ASTAR not only suffers from frequently synchronization, but also frequent squashes; For GOBMK and SJENG the performance improvement for TLS is negligible in all configurations.

To summarize, with our existing single-level TLS execution model, only a few benchmarks are able to scale with the number of cores; and even for the benchmarks that do scale, most of them scale sub-linearly. While the reasons for the lack of scalability differ from benchmark to benchmark, it is obvious that the amount of parallelism is limited.

7.2.2 Factors affecting scalability:

We show the breakdown of execution time for SPEC 2006 in Fig. 7.2 (same as shown in chapter 3, Fig. 5.2). We can see that in many benchmarks, significant portion of the execution time of TLS is wasted due to mis-speculations and idling of cores caused by synchronization and lack of threads. Several architecture and compiler techniques have been developed to reduce the impact of these TLS overheads. In spite of this, the performance of TLS has been limited.

Several program characteristics could limit the TLS performance of loops:

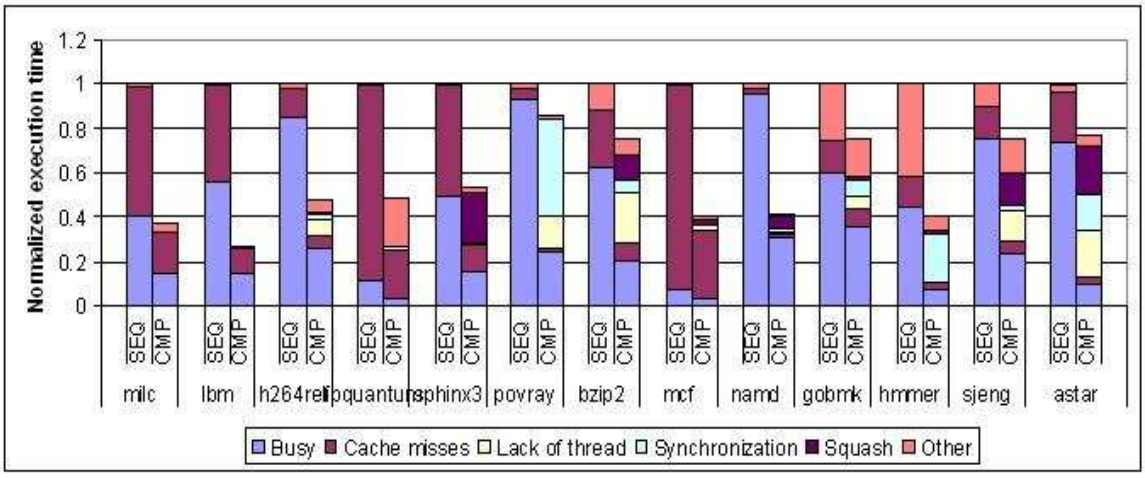


Figure 7.2: Shows the breakdown of execution time while executing the selected loops normalized to sequential execution time

Data-dependence violations

Since the loops selected for single-level TLS are based on accurate performance estimation, it will not have too many squashes. But the probability of a thread being squashed due to data-dependence violation increases as the number of threads increase. Say, the probability of a inter-thread dependence occurring between iteration *i* and *i-1* is P_i. The probability of a thread getting squashed due to inter thread dependence is P_i. But this thread (*i*) can also be squashed if any of its predecessor threads get squashed. So the actual probability of squash is $P_1.P_2.P_3$ ... P_i. Hence if the number of active threads increase, the probability of mis-speculation also increases.

Also some loops could have inter-iteration dependence with distance > 1. If the dependence

distance is *d*, the loops will not have any squash if there are at most *d*-1 active threads. But the *d*th thread will have a dependence with the first thread. For such loops, increasing the number of threads increases mis-speculations with no improvement in speedup.

Control violations (Iteration count)

When the loop iteration count is small, the number of iterations yet to be executed could be smaller than the number of available cores. This leads to idle cores or thread violations if the loop ”breaks” while executing an iteration. The same performance can be extracted by using a smaller number of cores.

Synchronization delay

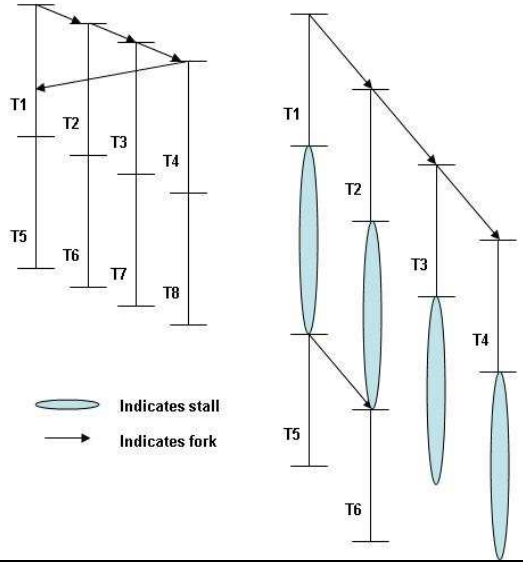


Figure 7.3: Effect of synchronization

In our TLS model, inter-iteration register dependences and inter-iteration memory dependences that always happen are synchronized to avoid frequent mis-speculations. Let d be the synchronization delay and S be the average execution time of the iteration. As shown in Fig. 7.3, there are two cases on how the delay d affects the performance. In case 1, $D \leq S/p$, where p (p=4 in Fig 7.3) is the number of processors. Here, after the first set of forks, all the cores are always busy executing the iterations till loop termination. In case 2, $D > S/p$, the large delay causes some cores to be idle waiting for value from the previous iteration. In case 1, there is minimal impact on performance due to synchronization delay but in case 2, synchronization limits performance. Here the same performance could be obtained by using a smaller number of cores.

Apart from these three major bottlenecks other factors like cache performance, speculative buffer overflow can limit the performance of single level TLS. Such loops with limited TLS performance cannot efficiently use all the available cores. In these cases, we could compliment these loops by simultaneously executing its inner or outer loops.

7.3 Multi-level TLS loop scheduling

One of the key challenges in supporting multi-level TLS is to allocate the available cores to the speculative threads from different loop levels to maximize overall performance. In this section we discuss in detail our compiler based loop scheduling technique.

7.3.1 Static vs dynamic loop selection

For multi-level TLS, the loop selection process, in addition to selecting a set of loops, should also assign specific number of cores to each loop. Without such core allocation the loop which executes first (outermost loop) could monopolize all the available cores and thus preventing the later inner loops from forking any threads.

Renau *et. al* [47] proposed an architectural technique referred to as *Dynamic Task Merging*. Hardware counters are used to identify threads that suffer frequent squashes; and these threads are prohibited from spawning new threads. Other than requiring complex hardware support, the proposed approach has the following limitations. First of all, using number of squashes as a measurement of TLS efficiency is inaccurate. As we have seen in Section 7.2, the performance of TLS loops is also determined by synchronization, iteration count, load balancedness, and etc. Secondly, the hardware is unable to pre-determine the TLS potential of inner loops. As long as the outer loop does not show performance degradation, it will monopolize the cores. The inner loops will never be attempted for parallelization even if it has more parallelism. This will lead to a suboptimal allocation of cores.

On the other hand, the compiler has full knowledge of the loop nesting structure of the program. With the knowledge of estimated performance of each loop level, the compiler can potentially allocate the available cores to the different loop levels to extract near optimal performance.

7.3.2 Predicting performance for each loop

The first step in static compiler based loop selection is to predict the performance of each loop. In the case of a DOACROSS loop, the time required for the parallel execution can be calculated by its initiation delay *d* as shown in [80]. A similar method to estimate the performance of a TLS loop has been studied in [25, 48]. In this study, we use a simple simulator model to measure the performance of loops using *train* input set.

The model simulates an eight core in-order processor with rudimentary TLS support and each instruction is assumed to take 1 cycle. The model measures the important TLS overhead like cycles wasted in mis-speculations, cycles where the cores are idle (due to low iteration count), synchronization overhead, etc. To reduce simulation time, only the first 500 invocations of the loop are simulated.

For multi-level loop selection, we also need the performance of loops when smaller number of cores were used. To avoid re-simulating the loops, the simulation model derives the various overheads for smaller number of cores while simulating for eight-cores. For example, if the inter-iteration dependence causing violation has dependence distance of more than *x* then the violation would not have happened if we had used less than *x* number of cores. So, the resulting overhead due to violation will not be included in calculating the performance for using less than *x* number of cores. Similarly overheads due to synchronization delay and low iteration count are calculated for smaller number of cores. We validated the estimated performance using the simplified model with the simulation results using our detailed simulator and found that the simplified model accurately predicted the relative performance of the different loops in loop

nests.

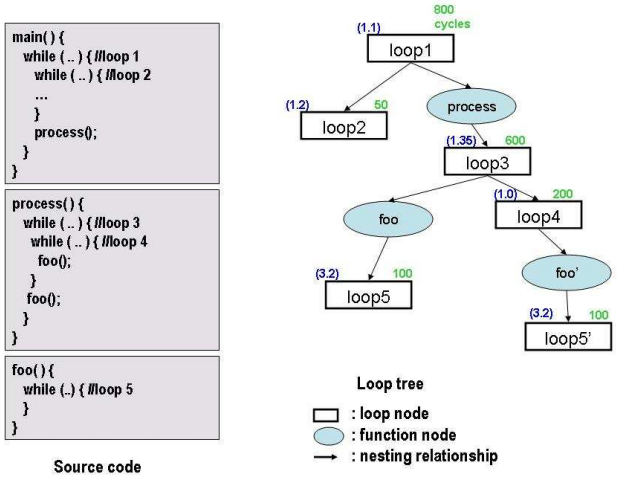
7.3.3 Loop selection for single-level TLS

Let $SingleSpeedup_{i,j}$ represent the estimated speedup achieved by parallelizing the loop $loop_i$ using j cores, where $j \in 2^0, 2^1, 2^2, \ldots 2^K$. This estimated per-loop speedup calculated is used for both single-level and multi-level loop selection. Our single-level loop selection algorithm is based on [25]. Let us consider a loop nest and its corresponding loop-tree [48] shown in Fig. 7.4(a).

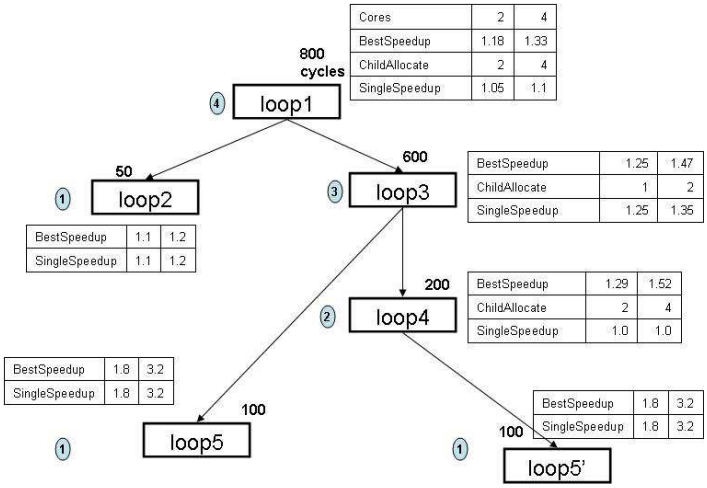
The number of cycles spent on each loop and the estimated speedup (using 2 and 4 cores) are shown for each loop node. The number of cycles spent on each loop is measured during the loop-tree instrumentation phase of our compiler (Chapter 2). The loop selection algorithm has to select a set of loops that don't overlap each other and also maximizes the total performance of the program. Candidate loop sets which do not cause loop overlap in Fig. 7.4(a) are {loop1}, {loop 2, loop 3}, {loop2, loop4, loop5}, {loop2, loop3} and {loop2, loop5, loop 5'}. Of these possible selections, {loop2, loop3} would give the maximum performance for the benchmark. This set would be selected by the compiler and parallelized.

In this example, the loop 3 has speedup of only 1.35 when using 4 cores and its speedup reduces to only 1.25 when using 2 cores. This indicates that loop 3 does not efficiently use all the 4 cores. So it is possible to allocate fewer cores to loop 3 without a large performance penalty and use the freed cores to parallelize loop 5 or loop 1.

With multi-level TLS we have more choices on how to use the available 4 cores – {(loop



(a) Example loop-tree



(b) Illustration of SpecOPTAL algorithm

Figure 7.4: Loop-tree based single-level and multi-level loop selection

3)} – allocate all 4 cores to loop3, {(loop 3), (loop 5, loop 5')} – allocate 2 cores to loop 3 and 2 cores to loop 5 and {(loop1),(loop3)} – allocate 2 cores to loop 1 and 2 cores to loop 3. Selecting the optimal allocation is NP-hard and we adapt a polynomial time, dynamic-programming based OPTAL algorithm [80] to allocate cores to maximize performance. The OPTAL algorithm [80] was originally used for loop allocation in nested DOACROSS and DOALL loops.

7.3.4 SpecOPTAL

The algorithm uses a dynamic programming based "bottom-up" approach to select the loops for multi-level TLS and to decide on the number of cores to allocate for each TLS loop, so that entire benchmark with multiple nested loops achieves the best performance. Due to the "bottom-up" nature of the algorithm, when allocating cores to a particular TLS loop level, only its immediate inner loops need to be considered.

The input to the algorithm is 2^K the maximum number of cores available for the benchmark. The algorithm operates on a loop tree generated during the loop-nest profiling phase of our compiler. The loop-tree (as shown in fig. 7.4(b)) is augmented with $SingleSpeedup_{i,j}$ and the execution time coverage of the loop. The output is the list of selected loops and the best core allocation for each TLS loop level. Let *BestSpeedup(i,j)* represent the best speedup achievable by parallelizing the multi-level TLS loop nest starting at ($loop_i$) using *j* cores.

Combining speedup

The basic step in the algorithm is to find the speedup of an outer loop when its inner loops are also parallelized. Lets call the function to calculate this as *GetCombinedSpeedup(L_i,M,N)*.

GetCombinedSpeedup(L_i,M,N) returns the speedup of loop L_i when we allocate M cores to parallelize L_i and N cores to parallelize its child loops in the loop tree $L_{i,j}$. *GetCombinedSpeedup(L_i,M,N)* is shown in Algorithm 2.

Recursive algorithm

The SpecOPTAL algorithm starts from the leaf level in the loop tree and calculates the speedup of parent nodes based on the child loops' speedup. The SpecOPTAL algorithm is shown in Algorithm 3. The exact allocation to the inner loops is given in the vector ChildAllocate(L_i,p). This would be used by the compiler to statically allocate cores. An example application of SpecOPTAL algorithm is shown in Fig. 7.4(b).

In the example in Fig. 7.4(b), at the first step the BestSpeedup(i,p) is calculated for loop5, loop5' and loop2. Loop4's BestSpeedup is calculated based on BestSpeedup of loop5'. The ChildAllocate for loop4 indicates that to achieve best performance for loop4, all of its allocated cores should be allocated to its child (loop5'). ChildAllocate of loop3 shows that to achieve best speedup using 4 cores, we need to allocate 2 cores to its children (loop5 and loop4). Finally the ChildAllocate of loop1 shows that it should allocate all its cores to its children (loop2 and loop3). So the best allocation of the loop nest in Fig. 7.4(b) is – loop2 can have 4 cores, loop3

can use only 2 cores and loop3's children loop5 and loop5' can use 2 cores.

Input: Outer loop L_i, M cores allocated to L_i, N cores allocated to the next level (L_i's child loops)

Output: Speedup of L_i with the specified allocation

Read Cycles(L_i) - the number of cycles spent in loop L_i from the profile;

foreach *Child of loop* L_i, $L_{i,j}$ **do**

/*Find total sequential cycles T_s for all inner loops.*/ Read Cycles($L_{i,j}$) - the number of cycles spent in loop $L_{i,j}$ when invoked from L_i from the profile;

SumBefPar += Cycles($L_{i,j}$);

/*Find total T_p for all inner loops after parallelization.*/ ParCycles($L_{i,j}$) = Cycles($L_{i,j}$) ÷ *BestSpeedup*($L_{i,j}$,*N*);

SumAfterPar += ParCycles($L_{i,j}$);

end

/*T_s of outer loop after inner loops are parallelized.*/

CyclesInnerPar = Cycles(L_i) - SumBefPar + SumAfterPar;

/*Calculate combined speedup.*/

ParCycles(L_i) = CyclesInnerPar ÷ $SingleSpeedup_{i,M}$;

return (Cycles(L_i) ÷ ParCycles(L_i)) ;

Algorithm 2: The *GetCombinedSpeedup*(L_i,*M*,*N*) to get the speedup of outer loop L_i when its inner loops are parallelized.

Input: Loop L_i and 2^K the number of cores to allocate.

Output: Estimated speedup of the entire benchmark - BestSpeedup(L_{root},2^K) and a vector ChildAllocate(L_i,p) which indicates for each loop, how many cores need to be allocated to its child loops.

if L_i *is* leaf **then**

  foreach p ∈ { 2^0, 2^1, 2^2, .. 2^K } **do**

    BestSpeedup(i,p) = SingleSpeedup$_{i,p}$;

  end

  return;

end

/*Allocate child loops first.*/ **foreach** $L_{i,j}$ *child of* L_i **do**

  SpecOPTAL($L_{i,j}$);

end

/*Inner loop's best allocation already known. Try all possible allocations for the outer loop*/ **foreach** p ∈ { *0,1,..K* } **do**

  foreach q ∈ { *0,1,..*p } **do**

    CurSpeedup(L_i,*q*) = *GetCombinedSpeedup*(L_i,2^q,$2^p/2^q$); **if** *CurSpeedup*(L_i,q) ≤ *MaxSpeedup* **then**

      MaxSpeedup = CurSpeedup(L_i,*q*); /*Record the best allocation*/

      ChildAllocate(L_i,p) = q;

    end

  end

  BestSpeedup(i,p) = MaxSpeedup;

end

Algorithm 3: The *SpecOPTAL algorithm.*

Complexity analysis

The SpecOPTAL is called for every node in the loop tree. Let λ be the number of nodes in the tree (including both loops and functions). In SpecOPTAL, the outer loop iterates for K times (2^K is the total number of cores) and the inner loop iterates on the average of K/2 times. So, total number of times GetCombinedSpeedup is called is $K^2/2$. The loop inside GetCombinedSpeedup iterates over all the children of the node. The average number of children per node is a constant (C_1) for a tree. Therefore, the total complexity of SpecOPTAL is O($\lambda K^2/2$).

7.4 SpecMerge architecture

In the previous section we saw how the SpecOPTAL efficiently allocates the available cores to the different loops. In this section we discuss the details our SpecMerge architecture which supports multi-level TLS in hardware.

7.4.1 Maintaining state of inner loops

In [47], Renau *e*t. al proposed a multi-versioned cache based architecture design to support out-of-order threads. Here each speculative thread is assigned an unique timestamp. The cache lines read or written by this speculative thread will be tagged with this unique timestamp and these cache lines are buffered in the cache till the speculative thread commits. If multiple threads access the same data, a new version is created for each thread, creating a multi-versioned cache [42]. In a multi-versioned cache all versions of the data are maintained in the same cache ”set”

of the set-associative cache. Thus the number of versions that can be maintained is limited by the associatively of the cache. For example, a 8-core processor with 4-way associative level-1 data cache can buffer 4*8= 32 unique versions.

Let us consider a 3-level nested loop:

```
for(i=0;i<X;i++) {// loop L1
   for(j=0;j<Y;j++) {// loop L2
    for(k=0;k<Z;k++) {// loop L3
   ...
   }}}
```

Let us assume that all three levels are TLS-parallelized. Say the outer loop creates one speculative thread. This outer-speculative thread (i=1) will create *Y* speculative threads from L2 and *Y***Z* inner speculative threads from L3. Also the non-speculative thread (i=0) will create *(Y-1)* L2 speculative threads and *(Y-1)***Z* + *Z-1* L3 speculative threads. If all these speculative threads access the same data in cache, we need *(Y-1)***Z* + *Z-1* + *(Y-1)* + *Y* + *Y***Z*. If we assume *Z==Y*, *Z* (the iteration count of the inner loops) can be at most 3. If the iteration count is larger, some threads need to be stalled and can also lead to preemptive squashing of later speculative threads. Due to the large number of versions that need to be maintained, this method is useful only when the iteration count of the inner loops are small.

Another disadvantage of multi-version cache is the increased cache access time. When a cache-set is accessed, in addition to address tags, the *TaskId* need to be compared to access the correct version. Also the method used in [47], requires hardware indirection tables to map real *TaskIds* to local *TaskIds*. This leads to increased hardware complexity and increased cache

access time.

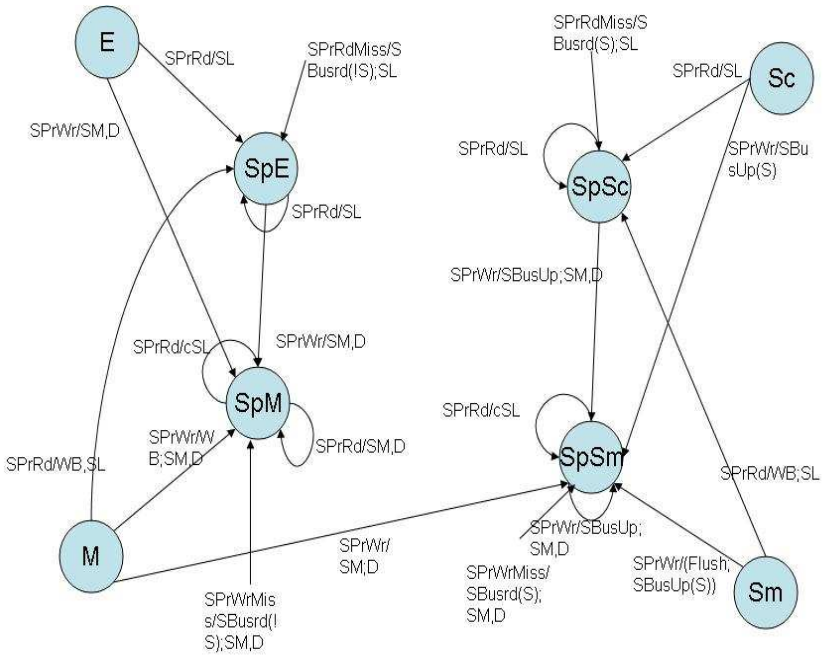
To counter the dis-advantages of this method, we propose *SpecMerge* architecture. Here when the speculative thread belonging to the inner loops complete, its speculative state is merged with the state of the outer loop iteration. So the total number of versions is limited by the number of speculative threads executing at a time, which is equal to the number of cores. Also since the number of versions is equal to the number of cores we can use a single-versioned cache and avoid the drawbacks due to multi-versioned cache.

Before describing the details of the SpecMerge architecture, we first briefly discuss the single-level TLS model used in this chapter.

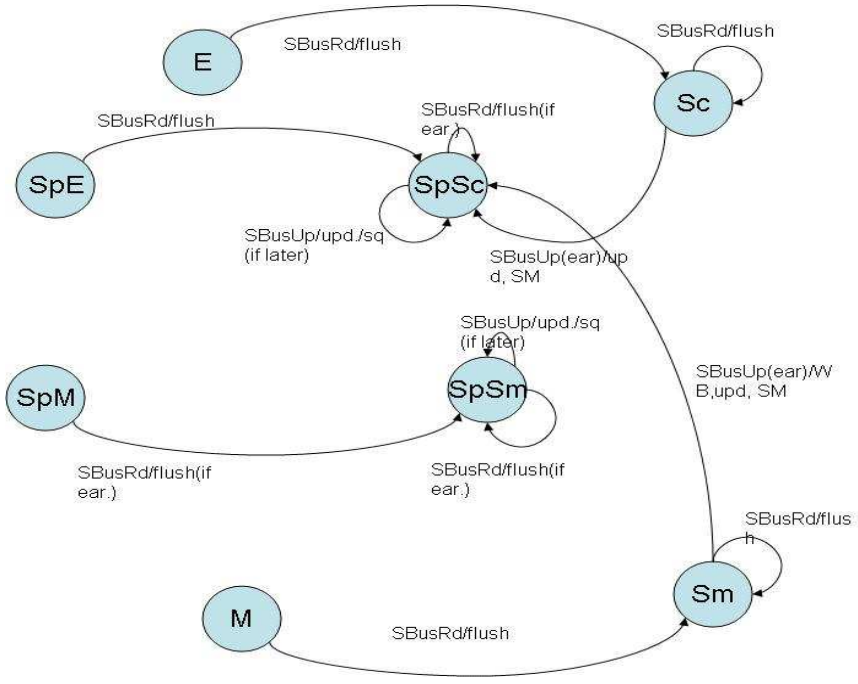
7.4.2 Single-level TLS model

In our TLS model as we saw in previous chapters, the threads are forked at the beginning of the iteration and any inter-iteration register dependences are communicated explicitly through special instructions as in [20]. Also the inter-iteration memory dependences that occur frequently are synchronized [63]. After inserting synchronization instructions instruction-scheduling is performed to increase the overlap between threads. This is in contrast with the model used in [47], where the fork is placed after the last inter-iteration dependence to avoid special handling of register dependences. Though the approach in [47] simplifies hardware, it sacrifices performance that can be gained through compiler optimizations are shown in [20, 48].

In contrast to previous chapters, in this chapter we extend the dragon protocol [81], an update-based cache coherence protocol to support TLS. Similar to [49], *SL* bits are used to track



(a) Cache transitions on processor requests.



(b) Cache transitions on bus requests.

Figure 7.5: Speculative dragon protocol to support single-level TLS.

Cache states	
E	Exclusive
M	Modified
Sc	shared-clean
Sm	shared-modified
SpE	Speculative Exclusive
SpM	Spec. Modified
SpSc	Spec. shared-clean
SpSm	Spec. shared-modified

Bus messages	
SBusUp	Spec. bus update
SBusRd	Spec. bus read
OverRd	Overflow read
OverWr	Overflow write
SBusUp(ear)	Spec. update from earlier thread

Processor commands	
SPrRd	Spec. read
SPrRdMiss	Spec. read that misses
SPrWr	Spec. write
SPrWrMiss	Spec. write that misses

Actions/effects	
SL,SM,D	set SL, SM, D bit
cSL	conditionally set SL bit (if !SM)
WB	write back data
flush	send the cache line to bus
upd.	update value
Sq.	Squash if current thread is "later"

Conditions	
S(!S)	Present (not present) in another cache
ear.	the thread is earlier in sequential order
later	the thread is later in sequential order

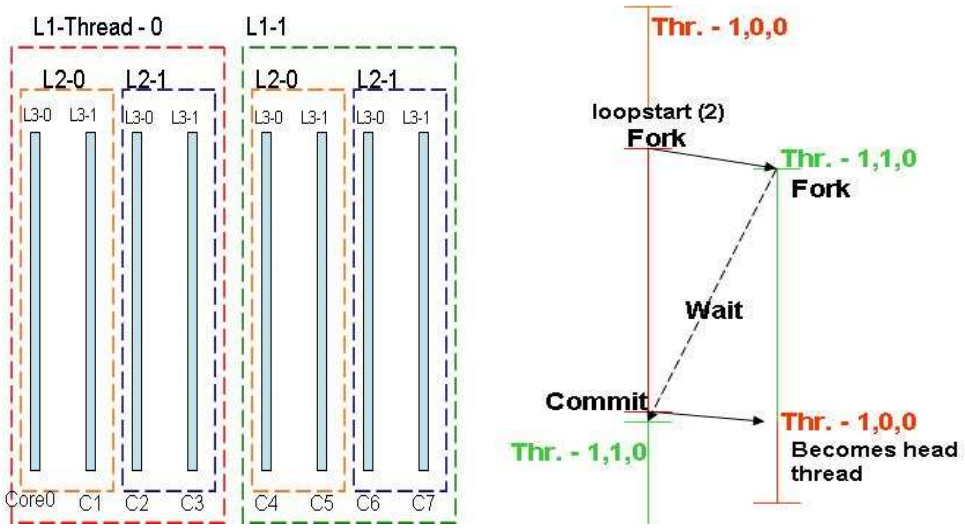
Figure 7.6: Description of cache states/coherence messages

speculative reads and *SM* bits are set to track speculative modifications. The cache transitions are shown in Fig. 7.5. For the sake of clarity, only the transitions related to TLS are shown. Explanation on the different cache states and messages are shown in Fig. 7.6

7.4.3 SpecMerge micro-architecture

Core allocation

The compiler inserts loop-start instruction at the beginning of each loop selected by the *SpecOP-TAL* algorithm. The number of cores allocated to the particular loop is passed as an operand. Say the loop is allocated *x* threads, and there are N cores available. An allocation of *x* implies that the loop can have utmost ”x” number of threads. We have *N* cores and the cores are equally partitioned among the threads, so each thread is allocated *N*/*x* cores. The allocated cores would



(a) Core allocation (b) Thread management

Figure 7.7: Allocation of cores and thread fork

be used exclusively by the thread and its descendant speculative threads created from inner loops. Say an inner loop is encountered with *y* as the allocated number of threads, the *N/x* available cores (allocated for the outer loop thread) are now partitioned so that each inner loop thread now gets *N/x*y* cores. This allocation of cores continues until each thread gets only one core. An example allocation for our triply nested loop is shown in Fig. 7.7(a).

As the loop allocation algorithm described in section 7.3, has full knowledge of the loop structure, all the loops that have been parallelized will get their allocated share of cores. In some cases due to recursion or difference between the profiled loop structure and the actual loop structure, an inner loop's request for cores may not be satisfied (all cores already allocated). In this case, the inner loop's request is ignored and no speculative threads are created from the inner loop.

TaskId

Let 2^k be the number of cores. Now k is the maximum number of loop levels that can be supported (each loop level would get 2 threads). The *TaskId* in our design is represented as a distance vector *(d_0,d_1,.. d_k)*, where d_i is the distance from the *head* thread in the loop level *i*. The *head* thread of a loop is the earliest thread in sequential order that is currently executing. Examples: If number of cores is 8, we could support 3 loop levels. The non-speculative thread would have a *TaskId* (0,0,0) as it is the earliest thread in sequential order. A thread with TaskId (0,1,2) is executing the earliest iteration in loop level 0 (outer most loop), 2nd earliest iteration in level 1 and 3rd earliest iteration in level 2. A TaskId (1,0,0) represents a thread executing the 2nd earliest iteration in level 0 and the earliest iteration in level 1 and level 2.

Relative sequential order between the threads can be inferred by comparing their corresponding distance vectors. A thread *X* with *TaskId (x_0,x_1,.. x_k)* is earlier than the thread *Y (y_0,y_1,.. y_k)*, if the following condition is satisfied:

$$\exists i \leq k | x_i < y_i \; and \; x_j = y_j \forall j < i \tag{7.3}$$

For example, the thread (0,1,2) is earlier than the thread (1,0,0) as it has a lesser distance at level 0.

Managing *TaskId*

TaskId for a thread is assigned during thread fork and changed only when there is a thread commit. When a thread forks a speculative thread, the next thread's *TaskId* is calculated based

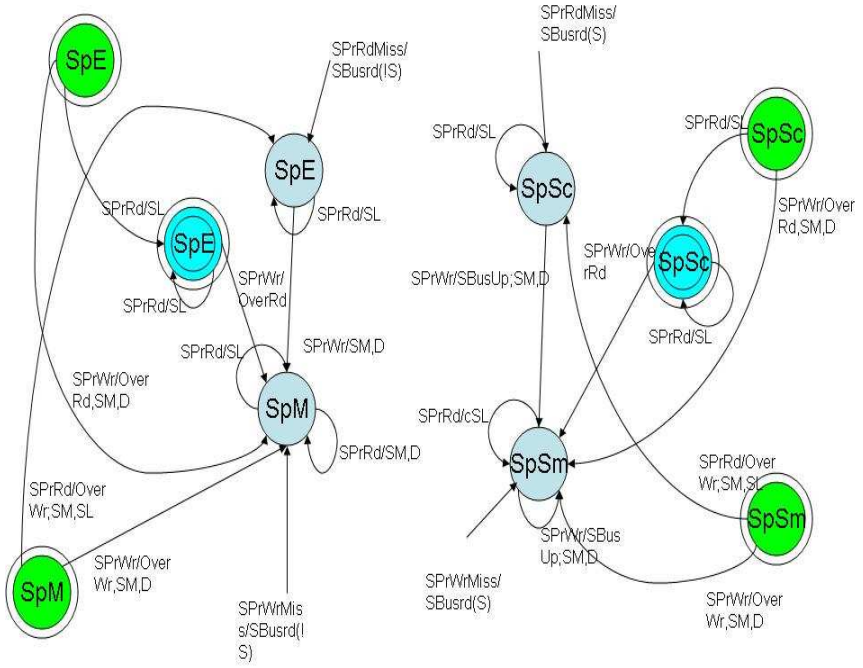
on the current thread's *TaskId* and loop level. If the current TaskId is (x_1,x_2,x_3) and if the loop level is 1 (second level), the next thread's TaskId would be (x_1,x_2+1,x_3). The next free core allocated to the current loop level is assigned to the new thread. If no free core is available, the speculative thread waits for the *head* thread to complete. An example is shown in Fig. 7.7(b).

When a thread commits, the immediate successor thread becomes the *head* thread of the loop level and its *TaskId* changes accordingly as shown in Fig. 7.7(b). Note that the commit in the example is not the same as the non-speculative commit, since the outer-loop thread (1,0,0) is still speculative. More details on commit operation will be discussed later.

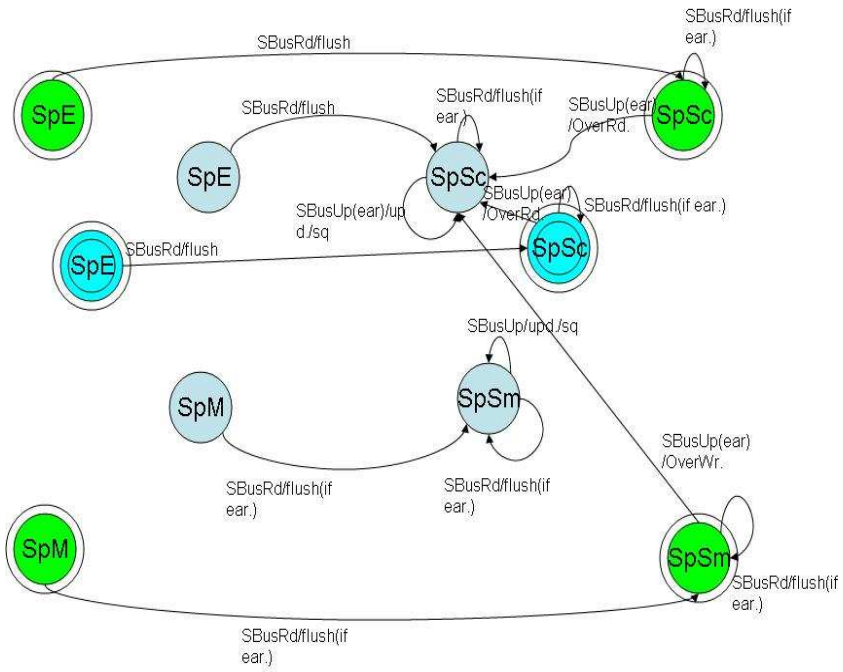
Speculative load

Like in [49], speculative loads are tracked using *SL* bits. As shown in Fig. 7.9(a), when the speculative thread (1,0,0) performs a load operation, the *SL* bit is set in the cache. When the inner loop iteration finishes execution it causes a commit, but unlike a non-speculative commit it does not clear the SL bit as there is still a possible dependence violation caused by a stores in earlier threads, eg. (0,0,0). The commit of (1,0,0) only indicates that the next thread (1,1,0) is the new *head* thread.

After the commit, a new speculative thread (1,1,0) from the inner loop (the original thread (1,1,0) is now the *head* thread with TaskId (1,0,0)) could be using the same core, as shown in Fig. 7.9(a) When the new speculative thread executes *load A*, it has to record this speculative load. If we have just one *SL* bit, we would not be able to differentiate between the speculative loads performed by thread (1,0,0) and by the thread (1,1,0).



(a) Cache transitions caused by processor requests.



(b) Cache transitions caused by bus requests.

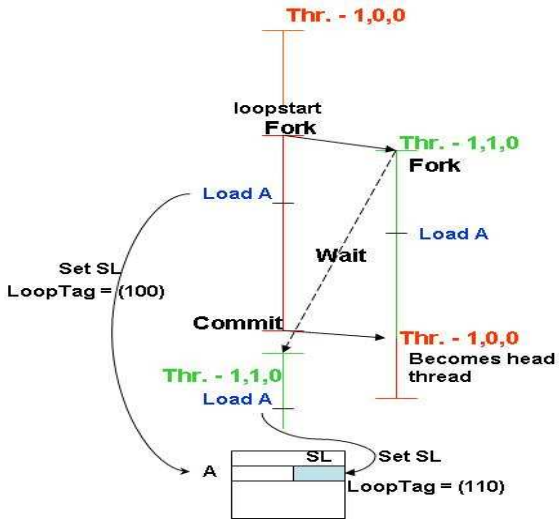
Figure 7.8: Modifications to support Multi-level TLS.

To differentiate between the speculative loads by the inner loops from the speculative loads of the outer loop, a *LoopTag* field is introduced. The *LoopTag* field uses *k* bits to support *k*-nested loops. (With 8 cores we can support 3 levels which would require a 3 bit LoopTag field). A speculative load from a particular loop level, would set the corresponding bit in the *LoopTag* as shown in fig. 7.9(a).

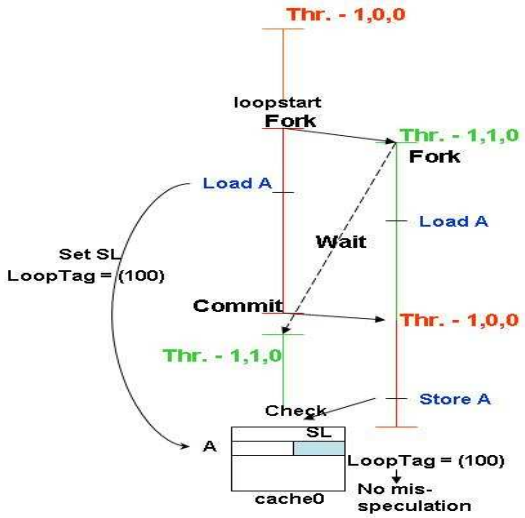
Fig. 7.8 shows all the cache state transitions due to speculative reads and speculative writes. For the sake of clarity, only the transitions related to multi-level TLS are shown. Double-circled state in Fig. 7.8 indicates that the cache line contains outer-thread's information (*LoopTag* indicate the loop level) and a triple-circled state indicates that the cache line contains both inner-thread and outer-thread's information.

Dependence checking

Dependence checking is done similar to [49] as shown in Fig. 7.8 and in Fig. 7.5. The relative order between between threads is determined by checking their distance vectors as shown before. In addition to *TaskIds* the *LoopTag* needs to be considered. Consider the example shown in Fig. 7.9(b). Store in thread (1,0,0) in core-1 causes a check of dependence in core-0. Core-0 has a cache line containing address *A* and its *TaskId* is (1,1,0) which makes it more speculative than the thread (1,0,0). But since the *LoopTag* (100) indicates that this line was read by an outer thread, no mis-speculation is caused.



(a) Speculative load.



(b) Dependence checking.

Speculative store

As in [49], *SM* bits are used to track speculative stores. As shown in Fig. 7.9(c), when thread (1,0,0) executes a store, the speculative value is buffered in the data cache, the *SM* bit is set and also the *LoopTag* is set. Let us consider the case when, after thread (1,0,0) commits and the next speculative (1,1,0) executing in the same core (core-0) executes a store to the same location *A*. This new value needs to be buffered, but the cache already contains a version of *A*. One option is to create a new version and tag the new version with *TaskId*. But this creates a multi-version cache and involves additional hardware cost (more bits to record *TaskId*). To avoid this we need to *replace* the old version of *A* belonging to thread (1,0,0). But this version of *A* is a speculative value and cannot be *written-back* to L2 cache.

From the Fig. 7.9(c), we see that when thread (1,0,0) commits in core0, the thread (1,1,0) in core1 becomes the *head* thread and its *TaskId* becomes (1,0,0). Since, core-1 holds the state for thread (1,0,0), we can transfer (*Overflow*) the version of *A* belonging to thread (1,0,0) from core-0 to core-1. This ensures that the speculative value of *A* from the thread (1,0,0) is buffered. Since such *Overflow* operations occur only during *store* operation, the additional delay caused can be easily tolerated using a write-buffer.

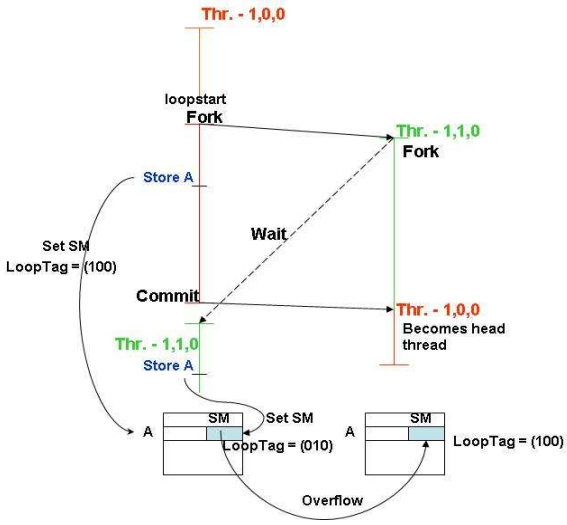
Lets consider a scenario when the thread (1,1,0) stores to the same location *A* that was speculatively read by thread (1,0,0) as shown in fig. 7.9(d). Let us assume that we use the same cache line to record this new value along with the *SL* bit set by (1,0,0). If the thread (1,1,0) mis-speculates, all its speculative cache lines would be invalidated. If we invalidate the cache line holding *A*, we would lose the *SL* bit set by (1,0,0). To avoid this we transfer (*Overflow*) the

cache line (along with *SL* bits) to core-1 which holds the current state of (1,0,0). After *Overflow* the *LoopTag* is cleared and the line would be used exclusively by the thread (1,1,0).

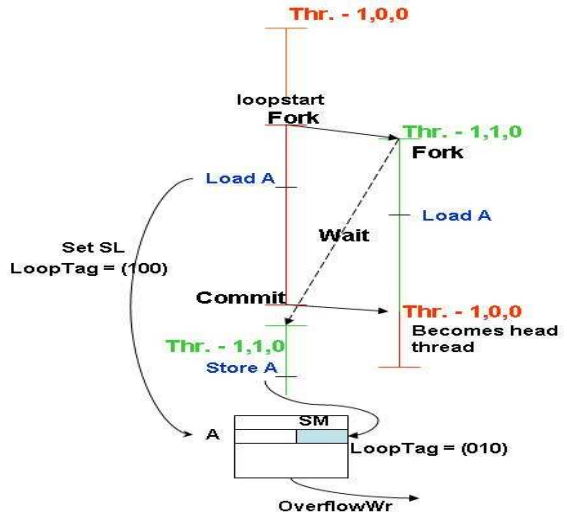
Commit:

In single-level TLS architectures, when a thread commits, the immediate next thread in made the non-speculative thread and all its speculative state in the cache is committed. In multi-level the state of the next thread could be spread on multiple caches. For example consider Fig. 7.10. The head thread (1,0,0) shifts between core-2 and core-3, consequently the state of the thread (1,0,0) is spread on cache-2 and cache-3. In the example, cache lines A,C and B belong to (1,0,0).So when the thread (0,0,0) finishes execution at the outer most loop, instead of sending the COMMIT token (”home-free” token in [49]) to one particular successor core, the token is sent to all cores containing state of (1,0,0). This is accomplished by a *prefix-compare* of the destination *TaskId* (1,0,0) – the threads (1,0,0) and (1,1,0) both have same prefix (1,0,0) corresponding to the outer-most loop. Both core-2 and core-3 would receive the COMMIT token from (0,0,0) and commit the state corresponding to (1,0,0).

Non-speculative commit occurs when the non-speculative thread finishes and sends the COMMIT token to its successors (Example shown in Fig. 7.10). The non-speculative commit operation is similar to [49], where on receiving the token all the speculative bits (*SL*, *SM*, *LoopTag*) are cleared. In Fig. 7.10 the *SL* bits and *LoopTag* of A, C and B are cleared (*conditional-gangclear*). Another type of commit operation called *Speculative Merge*, occurs when the *head* thread of a loop finishes (eg. (1,0,0)) and the next thread becomes *head* (eg.



(c) Scenario-1.



(d) Scenario-2.

Figure 7.9: Speculative store handling

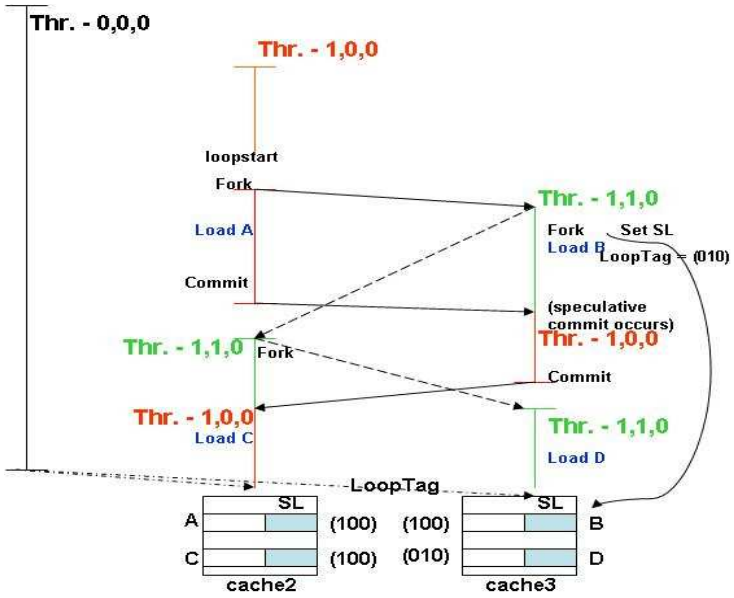


Figure 7.10: Commit operation

(1,1,0)). Now the state of the new *head* thread is merged with the outer-loop's thread. For example in Fig. 7.10, the *LoopTag* of B which was originally (010) is changed to (100) since it now belongs the outer-loop thread (1,0,0). This operation involves conditionally *gang-clearing* and *gang-setting* of *LoopTag* bits.

Hardware cost:

When compared to [47], the hardware cost of the SpecMerge scheme is minimal. Apart from the per cache line *LoopTag* bits (3 bits for 8 core CMP) no other additional hardware is needed when compared to single-level TLS. Other requirements like conditional *gang-clear*, *gang-set*, etc are already part of single-level TLS. Only significant cost in our scheme is the additional complexity in the cache controller logic (due to additional cache state transitions).

Table 7.1: Architectural parameters.

Parameter	
Fetch/Decode/Issue/Retire Width	6/6/4/4
Integer units	6 units / 1 cycle latency
Floating point units	4 units / 12 cycle latency
Memory ports	2Read, 1Write ports
Register Update Unit (ROB,issue queue)	128 entries
LSQ size	64 entries
L1I Cache	64K, 4 way 32B
L1D Cache	64K, 4 way 32B
Cache Latency	L1 1 cycle, L2 18 cycles
Memory latency	150 cycles for 1st chunk, 18 cycles subsequent chunks
Unifi ed L2	2MB, 8 way associative, 64B blocksize
Physical registers per thread	128 Integer, 128 Floating point and 64 predicate registers
Thread overhead	5 cycles fork, 5 cycles commit and 1 cycle inter-thread communication

7.5 Evaluation

In this chapter we use our experimental framework detailed in chapter 2. The exact processor configuration used is shown in Table 7.1.

7.5.1 Results

We applied our allocation algorithm to extract speculative threads for all benchmarks, but many benchmarks did not show any potential for multi-level TLS due to the following reasons: BZIP2, GOBMK and SJENG are omitted due to the lack of TLS parallelism overall; LIBQUANTUM and HUMMER are omitted due to the lack of nested parallelizable loops; LBM, NAMD, MILC, SPHINX3, H264REF and MCF are omitted due to the fact that the SpecOPTAL algorithm is unable to identify multiple levels of loops that can perform better than the single level TLS.

For benchmarks POVRAY and ASTAR the multi-level TLS showed good potential. We also

added results for MediaBench-2 benchmark to demonstrate the effectiveness of our multi-level TLS methodology.

7.5.2 Benchmarks

povray

POV-ray (SPEC 2006 - floating point) is a ray-tracing technique that calculates an image of a scene by simulating the way light rays travel in the real world. The benchmark iterates over all the pixels in the screen and sends out fixed number of rays for each pixel. When the rays intersect the different objects in the scene the color of the object is calculated. Apart from the outer loops that iterate over the pixels, all the major loops have very low iteration count. The median iteration count of the top 60 high coverage loops is *6*. The loops iterate over a fixed number of textures, number of entries in a light tree, number of intersections, etc and in each case the iteration count is very low. Due to this, the benchmark has only limited potential for single-level TLS. With multi-level TLS it is possible to simultaneously parallelize some of these low iteration count loops to better utilize the available cores.

jpegdec

JPEG 2000 (MediaBench-2 [1]) is a wavelet-based image compression standard. After initial color transformation, the image is split into *tiles*. All further operations are performed at the granularity of *tiles* and the loops that iterate over the different components in a tile have very

[1] http://euler.slu.edu/ fritts/mediabench/

low iteration count. The median iteration count of the top 60 loops in jpegdec is *8*. Similar low iteration count loops are common in many media and network applications that operate on fixed frame or fixed packet of data. In these cases single-level TLS would not be sufficient to extract all the available parallelism. With multi-level TLS some of these loops can be parallelized together to extract performance.

astar

Astar (SPEC 2006 - integer) is derived from a portable 2D path-finding library used in game AI. The library implements three different path finding algorithms. Unlike the other two benchmarks, in astar the median iteration count of the top 60 loops is around *186*. But the loops have large number of inter-thread data dependences that cause frequent mis-speculations. For some of the loops, frequent dependences are synchronized leading to large synchronization delay. Such frequent squash/synchronization behavior is common in many SPEC integer benchmarks, which limits their single level TLS performance. With multi-level TLS, we could reduce the cores allocated to these low performing loops and reallocate them to their inner or outer loops. With this combined parallelism it is possible to limit the wastage of resources due to mis-speculations and synchronization.

7.5.3 Results

Fig. 7.11(a) shows the speedup of both single-level and multi-level TLS over the sequential execution. The single-level TLS has a geometric mean performance of 45% when compared

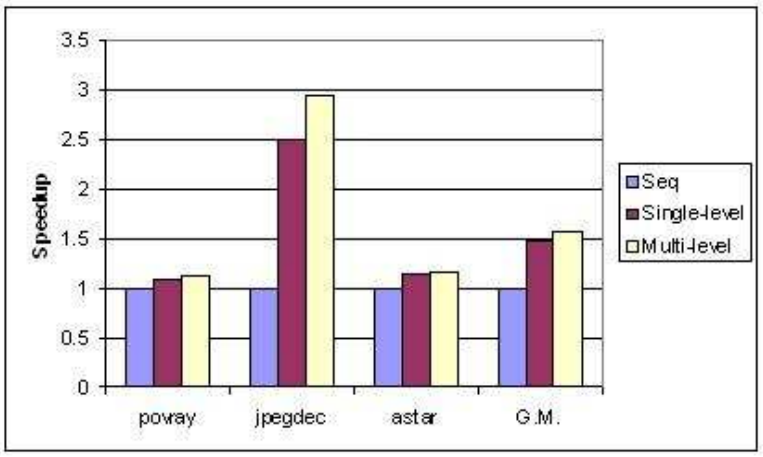
to sequential execution while multi-level TLS has a geometric mean performance of 57%. Fig. 7.11(b) shows the normalized execution time breakdown of the three different architectures (sequential, single-level TLS and multi-level TLS).

Discussion:

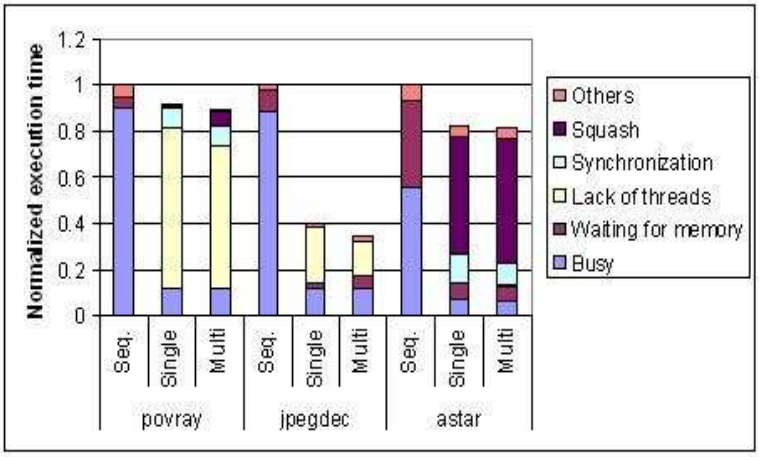
As shown in Fig. 7.11(b), in the benchmark *povray* as significant portion of execution time is spent in idling due to lack of threads. Single-level TLS selects the loop in file *csg.cpp* at *line 248* that has a coverage of 60% of the entire benchmark's execution time (code snippets for this loop were shown in the introduction). The average number of iterations per invocation of the loop is less than 4. This low iteration count leads to "idling" of cores and a limited speedup of only 9%. The loop in *line 248* has an inner loop at *line 258* whose average count is approximately 2. When this loop is also parallelized using multi-level TLS, the performance increases to 13%. As we can see, each loop at *line 248* and *line 258* cannot by themselves utilize all the available cores, but together they can lead to better performance. The limited improvement (4%) in performance is because, the inner loop forms only 16% of the execution of the outer loop and has a very low iteration count.

For benchmark *jpegdec* we see a large portion of execution time is wasted due to lack of threads (Fig. 7.11(b)). As we discussed, *jpegdec's* suffer from low iteration count. For example the single-level TLS selects the loop in file *pnm_enc.c*, *line 345*:

```
344:  for (y = 0; y < hdr->height; ++y) {
345:  for (cmptno = 0; cmptno < numcmpts; ++cmptno) { //Selected by single-level TLS
```



(a) Speedup



(b) Execution time breakdown

Figure 7.11: Comparing between single-level TLS and multi-level TLS

The average iteration count of this loop is only 3 (indicating three color components RGB), leading to idle cores. The outer loop in *line 344* suffers from large synchronization delay (inter-thread dependences) and also stalling due to speculative buffer overflow since the iteration size is large (about 1 Million instructions per iteration). Due to its limited performance, it was not selected for single-level TLS. With smaller number of cores, the impact due to synchronization delay can be reduced and with multi-level TLS the state of the outer loop is buffered on multiple cores (similar to example shown in Fig. 7.10) leading to reduced speculative buffer overflow effect. With mutli-level TLS, when both the loops in *line 344* and *line 345* are parallelized, we achieve a 43% increase in speedup.

In benchmark *astar* the loop selected for single-level TLS is in file *Way2_cpp* at *line 100*. As we have seen in Fig. 7.11(b), the single-level TLS suffers from large synchronization delay and frequent squashes. Due to the limited performance of loop at *line 100*, the cores are not effectively used. With multi-level TLS, an inner loop at *line 65* is also selected, which by itself has limited potential due to low iteration count (approx. 3). The combined performance of loops in *line 100* and *line 65* leads to an additional overall performance of 2% compared to single-level TLS. The low performance increase is because the inner loop also does not efficiently utilize the available cores as it also suffers from frequent squashes.

Impact on bus traffic:

According to our discussion in Section 7.4 the multi-level TLS could potentially cause an increase the amount of traffic in the common bus between cores in the CMP. The multi-level TLS

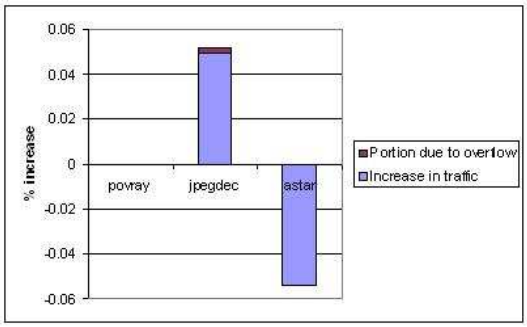


Figure 7.12: Increase in traffic in multi-level TLS

causes *Overflow* messages which where not present in single-level TLS. Also there can potentially be more sharing of data between threads in inner loops than in outer loops which can also lead to more traffic. On the other hand, if the single-level or multi-level causes more squashes it can also induce more traffic in the bus due to re-executions of the same thread.

Fig. 7.12 shows the percentage increase in the traffic of multi-level TLS when compared to single-level TLS. For benchmark *povray* there is almost no change in the amount of traffic. Here the average iteration count of the inner loop is 2 and the inner loop is only 16% of the outer-loop execution time. Due to this there is no significant increase in sharing of data between cores and consequently no increase in bus traffic. Similarly the benchmark *jpegdec* shows only 5% increase in traffic due to its small inner loop. In benchmark *astar* the single-level TLS suffers frequent squashes at the outer loop. Due to the frequent re-execution of the larger outer loop iteration, it causes more traffic than the multi-level TLS.

Impact of *Overflow* messages As described in Section 7.4, the *Overflow* messages occur in multi-level TLS which do not appear in single-level TLS. Fig. 7.12 shows the percentage of traffic increase in multi-level TLS that is due to the *Overflow* messages. As we discussed above,

there is no significant increase in sharing of data between threads which also leads to fewer *Overflow* messages.

Summary:

Our results clearly show that due to various benchmark characteristics - low iteration count (*povray,jpegdec*), synchronization delay (*astar, jpegdec*), mis-speculations (*astar*), the performance of single-level TLS can be limited. With multi-level TLS, more threads could be extracted from other loops (both inner and outer) leading to additional performance. Also we show that the *SpecMerge* architecture is efficient and does not lead to significant increase in bus traffic.

7.6 Conclusions

With increasing number of cores available, it is important to expose parallelism at multiple granularity to fully utilize all the available cores. To exploit speculative parallelism at multiple levels, two key challenges need to be addressed - resource allocation among threads and enforcing sequential commit order. Existing techniques use complex hardware monitoring to allocate cores and complex multi-versioned cache to implement multi-level speculative threads.

In this chapter we proposed *SpecOPTAL*, a novel compiler based static resource allocation scheme which allocates cores statically without need for complex hardware based monitoring schemes. Also we propose *SpecMerge* architecture which uses single versioned cache to support multi-level speculative threads. We show that our scheme can achieve a geometric mean

speedup of 15% over the single-level scheme on selected benchmarks. The SpecMerge scheme could potentially cause increase in bus traffic due to additional cache transitions that were not applicable for the single-level scheme. Our results show that the increase in traffic is only minimal showing the effectiveness of our scheme.

Chapter 8

Conclusion and Future Work

With the current trend towards multi-core/multi-threaded processors it is important to extract parallelism in applications to utilize these architectures to improve performance of programs. Thread Level Speculation (TLS) has been used to exploit parallelism in applications that are harder to extract using traditional compiler techniques due to ambiguous dependences. With different kinds of multi-threaded or multi-core design choices, it is important to understand the relative advantages of different architectures and develop techniques that can efficiently extract speculative parallelism.

In this thesis, we showed the performance potential for TLS in SPEC 2006 benchmarks to be about 60% compared to only 26% in SPEC 2000 benchmarks. By comparing the dependence behavior and performance characteristics of the more recent SPEC 2006 with the older SPEC 2000 benchmarks, we show a trend towards more parallel benchmarks which can benefit from speculation support. Given this trend, it is important to support efficient TLS architectures in

future multi-core/multithreaded processors.

We propose a novel cache-based architecture to support TLS in SMT processors. Unlike the previous approaches that use the Load-Store Queues (LSQ) which are fully-associative structures, we utilize the cache to support TLS. We show that for selected benchmarks that have larger threads, our cache based TLS approach can outperform the LSQ based approach by 19%.

We perform a detailed comparison of our SMT based TLS approach with the existing CMP based approach in terms of performance, power consumption, energy efficiency and thermal behavior. We show that the efficiency of TLS in each of these architectures depend on the characteristics of each benchmark. For programs that have limited potential for TLS, SMT based TLS is more efficient. While for the more parallel benchmarks, the CMP based TLS is shown to be more efficient.

To extract the TLS available in a more efficient way, we propose a SMT-CMP based *heterogeneous* multi-core architecture. We show that the potential of this approach is about 16% when compared to the best homogeneous configuration. We study the impact of switching overhead on energy efficiency and suggest potential ways to reduce the overhead.

One important challenge in future multi-core/multi-threaded architectures is to fully utilize all the available cores/threads. We propose compiler and architecture techniques that can exploit TLS parallelism at multiple levels. When compared to previous approaches that rely on complex hardware structures, we used compiler to allocate cores. We show that for selected benchmarks, our technique can achieve a speedup of 15% over single-level TLS.

8.1 Future work

The research presented in this thesis can be extended as follows:

- In this thesis, we showed the potential for improving efficiency by utilizing a *heterogeneous* multi-core architecture. Realizing the potential in *heterogeneous* multi-core architecture would involve developing techniques to minimize the overhead involved in switching between configurations. Also techniques need to be developed to predict the performance and power consumption of different loop regions at runtime.

- Even though we concentrated on TLS workloads, some of the techniques proposed can be utilized to improve efficiency in non-speculative parallel workloads.

- In this thesis, we concentrated on improving the efficiency of the execution of a single benchmark. When we consider a multi-programmed environment, each of the simultaneously executing benchmarks could have TLS threads. In such a scenario it is important to develop techniques that can efficiently share the available cores/threads in the processor among all the speculative threads from different applications to improve the overall efficiency of the entire workload currently executing.

Chapter 9

References

[1] J. Emer. EV8: The Post-ultimate *Alpha*.(Keynote address). In *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[2] Intel Corportation. Intel Pentium 4 Processor with HT Technology. http://www.intel.com/personal/products /pentium4/hyperthreading.htm.

[3] Intel Corporation. Intel's Dual-Core Processor for Desktop PCs. http://www.intel.com/personal/desktopcomputer/dual_core/, 2005.

[4] AMD Corporation. Leading the Industry: Multi-core Technology & Dual-Core Processors from AMD. http://multicore.amd.com/en/Technology/, 2005.

[5] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, and T. Yamazaki Y. Watanabe. A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor. In *Hot Chips 17*, August 2005.

[6] T. Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 500–519, August 1986.

[7] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, December 1998.

[8] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 58–67, May 1992.

[9] M. Cintra and J. Torrellas. Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In *8th International Symposium on High-Performance Computer Architecture (HPCA-8)*, 2002.

[10] Pradeep Dubey, Kelvin O'Brien, Kathryn O'Brien, and Charles Barton. Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1995)*, June 1995.

[11] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 414–425, June 1995.

[12] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *Supercomputing '98*, November 1998.

[13] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of ASPLOS-VIII*, October 1998.

[14] V. Krishnan and J. Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 1999)*, October 1999.

[15] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *13th Annual ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[16] J. Oplinger, D. Heine, and M. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings PACT 99*, October 1999.

[17] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *27th Annual International Symposium on Computer Architecture (ISCA '00)*, June 2000.

[18] J.-Y. Tsai, J. Huang, C. Amlo, D.J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures*, 48(9), September 1999.

[19] Anasua Bhowmik and Manoj Franklin. A Fast Approximate Interprocedural Analysis for Speculative Multithreading Compiler. In *17th Annual ACM International Conference on Supercomputing*, 2003.

[20] A. Zhai, C. B. Colohan, J. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct 2002.

[21] Antonia Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Memory-Resident Value Communication Between Speculative Threads. In *The 2004 International Symposium on Code Generation and Optimization*, Mar 2004.

[22] 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 *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)*, June 2004.

[23] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *ACM SIGPLAN 04 Conference on Programming Language Design and Implementation (PLDI'04)*, June 2004.

[24] T. N. Vijaykumar and Gurindar S. Sohi. Task Selection for a Multiscalar Processor. In *31st Annual IEEE/ACM International Symposium on Microarchitecture (Micro-31)*, November 1998.

[25] S. Wang, K. S. Yellajyosula, A. Zhai, and P.-C. Yew. Loop Selection for Thread-Level Speculation. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2005.

[26] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, March 2006.

[27] Lin Gao, Lian Li, Jingling Xue, and Tin-Fook Ngai. Exploiting speculative tlp in recursive programs by dynamic thread prediction. In *Compiler Construction, 13th International Conference, CC*, March 2009.

[28] Lin Gao 0002, Quan Hoang Nguyen, Lian Li 0002, Jingling Xue, and Tin-Fook Ngai. Thread-sensitive modulo scheduling for multicore processors. In *the Intl. Conference on Parallel Processing (ICPP)*, 2008.

[29] Lin Gao 0002, Lian Li 0002, Jingling Xue, and Tin-Fook Ngai. Loop recreation for thread-level speculation. In *13th International Conference on Parallel and Distributed Systems (ICPADS)*, 2007.

[30] I. Park, B. Falsafi, and T.N. Vijaykumar. Implicitly-multithreaded processors. In *30th Annual International Symposium on Computer Architecture (ISCA '03)*, June 2003.

[31] Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm, and Dean Tullsen. Converting Thread-Level Parallelism Into Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Computing Surveys*, pages 322–354, August 1997.

[32] Stefanos Kaxiras, Girija J. Narlikar, Alan D. Berenbaum, and Zhigang Hu. Comparing power consumption of an smt and a cmp dsp for mobile phone workloads. In *CASES*,

2001.

[33] Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The energy efficiency of cmp vs. smt for multimedia workloads. In *18th Annual ACM International Conference on Supercomputing*, pages 196–206, 2004.

[34] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, energy, and thermal considerations for smt and cmp architectures. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[35] James Burns and Jean-Luc Gaudiot. Area and system clock effects on smt/cmp throughput. *IEEE Trans. Computers*, 54(2):141–152, 2005.

[36] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[37] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime Compilation Methods for Multicomputers. In *International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.

[38] Cosmin E. Oancea and Alan Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, New York, NY, USA, 2008. ACM.

[39] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2003. ACM.

[40] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The r-lrpd test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 318, Washington, DC, USA, 2002. IEEE Computer Society.

[41] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5), May 1996.

[42] T.N. Vijaykumar, S. Gopal, J.E. Smith, and G. Sohi. Speculative versioning cache. In *IEEE Transactions on Parallel and Distributed Systems*, volume 12, pages 1305–1317, December 2001.

[43] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti R. Sarangi, James Tuck, and Josep Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26(1):80–91, 2006.

[44] J.Chen and L.K. John. Energy-aware application scheduling on a heterogeneous multicore system. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*, 2008.

[45] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, New York, NY, USA, 2006. ACM.

[46] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32(2), 2004.

[47] 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 *19th Annual ACM International Conference on Supercomputing*, June 2005.

[48] S. Wang. *Compiler Techniques for Thread-Level Speculation*. PhD thesis, University of Minnesota, 2007.

[49] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. In *ACM Trans. on Computer System*, volume 23, pages 253–300, August 2005.

[50] Open64 the open research compiler. http://www.open64.net/.

[51] Joshua J. Yi, Sreekumar V. Kodakara, Resit Sendag, David J. Lilja, and Douglas M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[52] E. Perelman, M. Polito, B. Calder, J. Sampson, J. Y. Bouguet, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *the 20th International Parallel and Distributed Processing Symposium*, April 2006.

[53] C-K Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN 05 Conference on Programming Language Design and Implementation (PLDI'05)*, June 2005.

[54] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, June 1997.

[55] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture (ISCA '00)*, 2000.

[56] H. Wang. Orion: A power-performance simulator for interconnection networks, 2002.

[57] W. Huang, K. Sankaranarayanan, R. J. Ribando, M. R. Stan, and Kevin Skadron. An improved block-based thermal model in hotspot 4.0 with granularity considerations. In *Workshop on Duplicating, Deconstructing, and Debunking, in conjunction with the 34th International Symposium on Computer Architecture (ISCA)*, 2007.

[58] Standard Performance Evaluation Corporation. The SPEC CPU 2006 Benchmark Suite. http://www.specbench.org.

[59] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In *ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming*, 2007.

[60] Standard Performance Evaluation Corporation. The SPEC CPU 2000 Benchmark Suite. http://www.specbench.org.

[61] F. Warg and P. om. Limits on speculative module-level parallelism in imperative and object-oriented programs on cmp platforms. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*.

[62] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *20th Annual ACM International Conference on Supercomputing*, 2006.

[63] S. Wang, A. Zhai, and P.-C. Yew. Exploiting Speculative Thread-Level Parallelism in Data Compression Applications. In *The 19th International Workshop on Languages and Compilers for Parallel Computing*, Oct 2006.

[64] Intel c++ compiler. http://www.intel.com/cd/software/products/asmo-na/eng/277618.htm.

[65] P. Marcuello and A. Gonzalez. Exploiting speculative thread-level parallelism on a smt processor. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, April 1999.

[66] V. Krishnan and J. Torrellas. A Chip Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture*, September 1999.

[67] Pedro Marcuello and Antonio González. Exploiting speculative thread-level parallelism on a smt processor. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 754–763, London, UK, 1999. Springer-Verlag.

[68] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Trans. on Computer System*, 23, Aug 2005.

[69] Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. Hardware Support for Large Speculative Threads. In *33rd Annual International Symposium on Computer Architecture (ISCA '06)*, Jun 2006.

[70] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Maximizing On-Chip Parallelism. In *22nd Annual International Symposium on Computer Architecture (ISCA '95)*, June 1995.

[71] Jose F. Martnez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellass. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In *35th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-35)*, Istanbul, Turkey, 2002.

[72] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, pages 364–373, 1990.

[73] V. Packirisamy, S. Wang, A.Zhai, W-C Hsu, and P-C Yew. Supporting speculative multi-threading on simultaneous multithreaded processors. In *12th International Conference on High Performance Computing HiPC'2006*, Bengaluru. India, December 2006.

[74] Fredrik Warg and Per Stenström. Dual-thread speculation: Two threads in the machine are worth eight in the bush. In *SBAC-PAD '06: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing*, pages 91–98, 2006.

[75] J.Donald and M.Martonosi. Temperature-aware design issues for smt and cmp architectures. In *Fifth Workshop on Complexity-Effective Design (WCED) in conjunction with ISCA-31*, June 2004.

[76] Yingmin Li, Lee B., David Brooks, Zhigang Hu, and Kevin Skadron. Cmp design space exploration subject to physical constraints. In *12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006.

[77] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: a power/performance/thermal view. In *20th ACM International Conference on Supercomputing (ICS'06)*, June, 2006.

[78] S. Marc, K. Reiner, L.P. Josep L., U. Theo, and V. Mateo. Transistor count and chip-space estimation of simplescalar-based microprocessor models. In *Workshop on Complexity-Effective Design, in conjunction with the 28th International Symposium on Computer Architecture*, June 2001.

[79] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*.

[80] Constantine D. Polychronopoulos, David J. Kuck, and David A. Padua. Utilizing multidimensional loop parallelism on large-scale parallel processor systems. *IEEE Trans. Computers*, 38(9), 1989.

[81] D.E.Culler, J.P.Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/software Approach*. The Morgan Kaufman Series in Computer Architecture and Design, San Francisco, CA, 1999.

Appendix A

List of loops parallelized

In this section, we describe the list of loops that were parallelized in this thesis. For each benchmark, we show the list of high coverage loops (atleast 5% dynamic loop coverage) that showed good parallelism. For each loop we provide the following information:

- Source file name and line number;
- Coverage of the loop;
- Speedup achieved using four core CMP;
- Average iteration size;
- Average iterations per invocation;
- A brief description of the loop.

Table A.1: Loops parallelized for 401.bzip2 benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
blocksort.c,551	14%	1.16	4880	2	Suffers from low iteration count and frequent squashes

Table A.2: Loops parallelized for 429.mcf benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
pbeampp.c,165	65%	2.40	335	292	Achieves good speedup. Needed some synchronization.

Table A.3: Loops parallelized for 433.milc benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
quark_stuff.c, 1523	33%	3.55	1972	20736	Loop is parallel.

Table A.4: Loops parallelized for 444.namd benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
ComputeNonbondedUtil.h, (calc_ pair_energy_merge_ fullelect)	4%	3.17	375	55	suffers few squashes.
ComputeNonbondedUtil.h, (calc_ pair_ nonbonded)	4%	3.09	63	230	Loop is parallel.

Table A.5: Loops parallelized for 429.povray benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
csg.cpp,248	60%	1.17	2458	4	Suffers from low iteration count.

Table A.6: Loops parallelized for 456.hmmer benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
fast_algorithms.c,133	78%	2.51	156	300	Large speedup due to speculative instruction scheduling.

Table A.7: Loops parallelized for 462.libquantum benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
gates.c,89	62%	1.18	43	32765	Loop is parallel, but iteration size is small.

Table A.8: Loops parallelized for 464.h264ref benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
mv-search.c,982	15%	3.74	68	1089	Loop has small iteration size.

Table A.9: Loops parallelized for 470.lbm benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
lbm.c,186	99%	1.19	525	1300000	Loop is parallel.

Table A.10: Loops parallelized for 473.astar benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
Way2_cpp,100	60%	1.17	1548	553	Need synchronization, suffers from squashes.

Table A.11: Loops parallelized for 482.sphinx3 benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
vector.c,513	37%	3.89	1108	2048	Loop is parallel.
approx_cont_mgau.c,27	36%	1.27	2189	6144	Frequent squashes but some prefetching effect.

Table A.12: Loops parallelized for 445.gobmk benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
reading.c,4171	13%	1.48	1472	3	Suffers from low iteration count.

Table A.13: Loops parallelized for 458.sjeng benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
search.c,199	13%	1.09	1268	7	Suffers from misspeculations and low iteration count.

Table A.14: Loops parallelized for 175.vpr(place) benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
place.c,897	60%	2.19	476	9	Needs synchronization.

Table A.15: Loops parallelized for 175.vpr(route) benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
route.c,777	48%	1.39	440	7	Needs synchronization.
route.c,1081	25%	1.36	58	12	Needs synchronization.

Table A.16: Loops parallelized for 176.gcc benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
flow.c,1422	12%	1.07	19	32	Suffers from small iteration size.

Table A.17: Loops parallelized for 177.mesa benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
texture.c,704	32%	1.29	557	2	Suffers from low iteration count.

Table A.18: Loops parallelized for 179.art benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
scanner.c,611	36%	1.09	51	11	Suffers from small iteration size.
scanner.c,584	29%	2.81	442	1000	Loop is parallel.

Table A.19: Loops parallelized for 183.equake benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
quake.c,462	37%	1.05	1429	30169	Needs synchronization.
quake.c,1204	43%	2.29	273	6	Loop is parallel.

Table A.20: Loops parallelized for 188.ammp benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
rectmm.c,562	82%	1.47	3361	336	Suffers from frequent mis-speculations.

Table A.21: Loops parallelized for 300.twolf benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
dimbox.c,82	7%	1.16	431	3	Suffers from mis-speculations and low iteration count.
dimbox.c,211	8%	2.59	581	3	Suffers from low iteration count.
dimbox.c,254	7%	2.57	539	3	Suffers from low iteration count.

Table A.22: Loops parallelized for 255.vortex benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
oa1.c,428	13%	1.05	11778	13	Suffers from misspeculations.

Table A.23: Loops parallelized for 253.perlbmk benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
regexec.c,1306	26%	1.2	708	2	Suffers from low iteration count.

Table A.24: Loops parallelized for 197.parser benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
parse.c,582	18%	1.02	3445644	8	Suffers from misspeculations and low iteration count.
main.c,1605	12%	1.01	1126562	21	Suffers from misspeculations.
prune.c,1126	2%	1.21	77	227	Suffers from misspeculations.

Table A.25: Loops parallelized for 181.mcf benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
implicit.c,228	47%	1.01	2776905	16555	Needs synchronization.
mcfutil.c,78	23%	2.53	1114	13384	Large benefit from prefetching.
pbeampp.c,181	20%	2.33	504	289	Benefits from prefetching, suffers from misspeculations.

Table A.26: Loops parallelized for 256.bzip2 benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
bzip2.c,1455	17%	1.14	22	50	Suffers from misspeculations.
bzip2.c,2047	11%	1.91	5705	28	Suffers from misspeculations.

Table A.27: Loops parallelized for 164.gzip benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
deflate.c,678	51%	1.04	2246	14713884	Suffers from misspeculations.
deflate.c,570	31%	1.08	1491	10128343	Suffers from misspeculations.

Table A.28: Loops parallelized for 186.crafty benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
quiesce.c,70	10%	3.04	793	4	Loop is parallel.

Table A.29: Loops parallelized for 254.gap benchmark.

Loop	Loop coverage	Speedup	Average iter. Size	Aver. Iters. per Invocation	Comments
gasman.c,772	23%	2.14	184	991182	Suffers from misspeculations.