Exploiting Parallelism in Multicore Processors through Dynamic Optimizations

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Yangchun Luo

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Prof. Antonia Zhai

November, 2011

Acknowledgements

First of all, I would like to thank my advisor Prof. Antonia Zhai for her invaluable guidance throughout my Ph.D career. I have literally learned every bit of doing academic research from her. Her advice included not only technical guidance, but also how to think as an engineer/researcher. I have benefited significantly from her *iterative* approach in improving my technical writing and presentation, which I am able to apply to other aspects of my life as well. Over the last four years working with her, I have evolved from a fresh college graduate to an expert in computer architecture, capable for independent research. Words cannot express my deep gratitude towards her. Thank you Antonia!

I would also like to thank my thesis committee members - Prof. Jon Weissman, Prof. Anand Tripathi, and Prof. Sachin Sapatnekar for their insights and supports in helping me improve this dissertation. I also owe a lot to Prof. Wei-Chung Hsu and Prof. Pen-Chung Yew for their insightful guidance at various stages of my Ph.D career.

Furthermore, I would like to thank to my colleagues. Dr. Xiaotong Zhuang (IBM, now at Google) and Dr. Min Xu (SeaMicro) provided me with a great deal of help at my junior stage as a software engineer. My fellow lab mate Dr. Venkatesan Packirisamy (now at nVidia) gave me lots and lots of help on the simulation infrastructure. I would also want to express my gratitude to other colleagues I have interacted with: Tong Chen (IBM), Guojin He (now at MathWorks), Jagan Jayaraj, Pei-Hung Lin, Jin-Woo Jung, Wei-Lung Hung, Guoqiang Yang, Pingqiang Zhou, Jieming Yin, Vineeth Mekkat, Anup

Holey, Ragavendra Natarajan, and Zheng Liu for their support and encouragement.

Last, I am deeply indebted to my parents who raised me up with all they have and love me unconditionally. My deepest gratitude goes to Paul and Adrienne Volk who always lifted me up in their thoughts and prayers.

Abstract

Efficiently utilizing multi-core processors to improve their performance potentials demands extracting thread-level parallelism from the applications. Various novel and sophisticated execution models have been proposed to extract thread-level parallelism from sequential programs. One such execution model, Thread-Level Speculation (TLS), allows potentially dependent threads to execute speculatively in parallel.

However, TLS execution is inherently unpredictable, and consequently incorrect speculation could degrade performance and/or energy efficiency for the multi-core systems. To address these issues, this dissertation proposes *dynamic* optimizations that exploit the benefit of successful speculations, while minimizing the impact of failed speculations.

First, we propose optimizations to *dynamically* determine where TLS should be applied in the original sequential program, whereas prior works have focused on using the compiler to statically select program regions. Our research shows that even the state-of-the-art compiler makes suboptimal decisions, due to the unpredictability of TLS execution. In this dissertation, speculative threads are monitored using the hardware-based counters and their performance impact is dynamically evaluated. Performance tuning policies are devised to adjust the behaviors of speculative threads accordingly. Dynamic performance tuning naturally allows the system to adapt to many program behaviors that are runtime dependent.

Second, we propose a heterogeneous multi-core architecture to support energy-efficient TLS. By carefully analyzing the behaviors of standard benchmark workloads, we identify a set of heterogeneous components that diversify in power and performance trade-offs and are also feasible to integrate. We have also devised a competent resource allocation scheme that *dynamically* monitors the program behavior, analyzes its characteristics, and matches it with the most energy-efficient configuration of the system.

Throttling mechanisms are introduced to mitigate the overhead associated with configuration changes. Under the context of TLS, our findings have shown that on-chip heterogeneity and dynamic resource allocation are two key ingredients for achieving performance improvement in an energy-efficient way.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

One primary research objective pertaining to computer system is to improve performance. In the past few decades, this objective had been largely realized through the efforts in increasing clock frequency and exploiting parallelism that exists among individual instructions. The exploration of the latter, also known as instruction-level parallelism (ILP), has led to many major innovations to computer architecture. For instance, on-chip cache component is integrated to reduce memory latency; the processor is pipelined so that multiple instructions can be executed simultaneously. Extensions such as branch prediction, out-of-order execution and many more are introduced to further exploit ILP and improve performance.

The pursuit along this path has hit a major stall since the beginning of the last decade. The power density issue has blocked the continuous increase in clock speed. As device size shrinks, cross-chip wire latency has made the highly inter-connected designs infeasible. Moreover, it has been observed that further exploiting ILP returns diminishing improvement. As a result, the computer industry has shifted from building a highly complex single-core processor to integrating multiple less complex cores on chip, resulting in a chip multi-processor (CMP).

The distributive nature of the CMP architecture is a realistic solution to issues

related to power density and cross-chip wire latency. However, improving performance for single-threaded programs is difficult. One way to achieve performance improvement on CMP is to break the sequential instruction stream into independent threads that feed into each core. In essence, efficiently utilizing multi-core processors demands thread-level parallelism (TLP) to be extracted from the applications.

Automatic thread extraction done by the compiler is desirable, but is challenging due to ambiguous memory aliasing and its effectiveness has been limited for general-purpose applications. In the presence of complex control flow and ambiguous pointer usage, traditional parallelization schemes must conservatively ensure correctness by synchronizing all potential dependences, which inevitably limits parallel performance. Another approach is to have programmers explicitly mark parallel sections and insert synchronizations. Unfortunately, this is time-consuming and error-prone.

Novel execution models have been proposed to extract thread-level parallelism with minimal manual intervention. These models include Thread-Level Speculation (TLS), Runahead Threads, Transactional Memory (TM). This dissertation focuses on the TLS execution model. TLS is a parallelization scheme that has attracted a large volume of research in the last decade. In the TLS model, the compiler is allowed to parallelize a sequential program without the burden to first prove the independence among the extracted threads. All potential dependences can be *speculatively* ignored at compile time. During runtime, dependence violation is detected and recovered by built-in hardware support. TLS execution can improve performance when dependence violation is infrequent. Using TLS, potential thread-level parallelism that is otherwise difficult to extract could be exploited without any manual effort in modifying the source code, since all the complexity is transparent from a programmer's perspective.

TLS execution has both advantages and disadvantages. When speculation is successful, it improves performance, and also reduces the duration in which static power is being consumed. However, when speculation fails, there may not be any performance benefit and dynamic power is wasted on speculative operations that eventually fail. Moreover,

supporting speculation means to keep more on-chip components alive, introducing more sources for static/leakage power. Thus, incorrect speculation can potentially degrade performance and/or energy efficiency for the multi-core systems.

1.1 Dissertation Objective and Summary

The objective of this dissertation is to propose optimizations that can take advantages of successful speculations, while minimizing the impact of failed speculations. Furthermore, we have learned that some of these optimization decisions are best made *dynamically* at runtime.

In the first half of this dissertation, we propose optimizations to *dynamically* determine where TLS should be applied to the original sequential program. Prior works have focused on using the compiler to statically select program regions for applying TLS. Our study shows that the unpredictability of TLS execution can exceed the capacity of even the state-of-the-art compiler-based analysis and result in suboptimal selections. Making the decision dynamically naturally allows the system to adapt to program behaviors that are runtime dependent.

In the second half of this dissertation, we propose a heterogeneous multi-core architecture to support energy-efficient TLS. By carefully analyzing the behaviors of standard benchmark workloads, we identify a set of heterogeneous components that diversify in power/performance trade-offs and are also feasible to integrate. We have also devised a competent resource allocation scheme that *dynamically* monitors the program behavior, analyzes its characteristics, and matches it with the most energy-efficient configuration of the system. Throttling mechanisms are introduced to mitigate the overhead associated with configuration changes. Our findings have shown that on-chip heterogeneity and dynamic resource allocation can enable achieving performance improvement in an energy-efficient way for TLS execution.

1.2 Related Work

This dissertation closely relates the following areas of computer architecture research: thread-level speculation (TLS), dynamic optimization, heterogeneous architecture design, and energy-efficient computation.

1.2.1 Thread-Level Speculation

A large body of research has been done in supporting TLS on a multi-threaded architecture [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. TLS can be supported on a multi-core chip (CMP) by extending the cache coherence protocol [11, 13]. It can also be supported using a Simultaneous Multi-Threading (SMT) processor [14]. For TLS to improve performance, one key issue is to decide how and where to extract speculative threads. Existing work fall into two categories: compiler-based [6, 15, 16, 17, 18, 19, 20, 21, 22] and hardware-based [23, 24]. Among compiler-based approaches, the POSH [21] TLS compiler partitioned the program into tasks based on code structures like loops and subroutines. It used a simple profiling pass for weeding out the ineffective tasks. Wang *et al.* [20] and Du *et al.* [17] used extensive profiling information to statically estimate loop performance and selected a set of loops for parallelization to maximize overall program performance. On the other hand, Marcuello *et al.* [24] and Tubella *et al.* [23] proposed hardware to dynamically detect loops and gathered characterization information. This information was then used to speculatively parallelize threads from a program.

The compiler-based approaches cannot predict many program behaviors accurately, because these behaviors often depend on numerous factors like micro-architecture-specific features, inputs applied, and memory access patterns, none of which can be projected beforehand. The hardware-based approaches cannot benefit from high-level information such as program structure and tend to only parallelize inner loops. In our hybrid mechanisms, compiler and hardware work collaboratively in tandem and we incorporate runtime analysis to determine where TLS should be applied.

1.2.2 Dynamic optimization

Dynamically detecting performance bottlenecks for program optimization has been demonstrated to be effective [25, 26, 27, 28, 29, 30]. For example, DynamoRIO [26, 28] used a combination of a native Just-In-Time compiler and partial evaluation techniques. Lu *et al.* [30, 29] generated helper thread prefetches using information obtained from the hardware monitors on the Sun UltraSPARC®. There is also a large body of work on runtime performance optimization for parallel applications (such as OpenMP [31]) [32, 33, 34, 35]. For example, Zhang *et al.* [35] experimented with different OpenMP scheduler configurations at different parallel regions. Lee *et al.* [34] peeled parallel loops and collected performance profiles using the first few iterations of the loop to re-optimize the program dynamically.

Our work is different because we focus on speculative threads, whose characteristics are quite different from the threads optimized in those frameworks. Moreover, we present a detailed methodology to evaluate the performance impact of speculative threads, whereas the other research works do not have such a quantitative evaluation.

1.2.3 Heterogeneous architecture design

Heterogeneous systems with same-ISA heterogeneous cores are proposed by Kumar *et al* to improve energy efficiency for sequential programs [36] and multi-programming workloads [37]. They presented policies for matching core sizes with application characteristics. Suleman *et al* [38] proposed Asymmetric Chip Multiprocessor to speed up lock-based critical section execution in OpenMP applications. Yang *et al* [39] proposed first-level cache designs that can be re-sized during program execution. Considerate energy-delay product reduction has been observed by having the ability of resizing both sets and associativities.

Our work is different because we have considered a large set of heterogeneous components that diversify in power and performance trade-offs. Our proposed architecture

has integrated both different cores and different cache sizes.

1.2.4 Energy-efficient computation

Various dynamic tuning mechanisms have been proposed for improving energy efficiency of microprocessors. Wu *et al* [40, 41] presented the runtime optimizer framework that uses dynamic voltage and frequency scaling (DVFS) to manage energy/performance trade-offs. Isci *et al* [42, 43] collected control flow, performance counters and live power measurement information from running applications, and found that performance-counter-based method consistently provided better representation of power behaviors. Bhattacharjee *et al* [44] proposed thread criticality predictor and then applied dynamic optimization and DVFS based on the criticality.

Although orthogonal, our work can potentially benefit from DVFS by having the ability to reduce the energy assumption on cores that are inefficiently used.

1.3 Dissertation Contributions

This dissertation has made several key contributions in optimizing both performance and energy efficiency for speculative thread execution on multi-core processors

1. This thesis proposes an execution framework that allows the runtime system to evaluate speculative thread execution and dynamically adjust their runtime behaviors.

2. This thesis proposes and evaluates dynamic performance evaluation methodologies that analyze execution cycle breakdown of speculative threads to determine their efficiency. It also discusses how hardware counters could be programmed to collect such cycle breakdown.

3. This thesis proposes, implements and evaluates various dynamic performance tuning policies that determine the priority in which the runtime system evaluates

speculative threads for parallel execution. By evaluating these policies, we identify important runtime information and compiler annotations that are key to optimizing speculative threads.

4. This thesis evaluates the energy efficiency of speculative threads on diverse architectural configurations; and identifies a subset that can enable energy-efficient execution for a large percentage of programs when integrated as a heterogeneous multi-core system.

5. This thesis identifies and evaluates various overheads associated with speculative thread migration and cache reconfiguration in a heterogeneous multi-core system; and proposes throttling techniques that can drastically mitigate such overheads.

6. This thesis proposes a dynamic resource allocation scheme that analyzes the characteristics of application with speculative threads and identifies an energy-efficient configuration on a heterogeneous multi-core system for each segment of execution.

1.4 Dissertation Outline

The organization of the rest of this dissertation is outlined as follows:

- Chapter 2 describes the evaluation infrastructure used in this thesis.

- Chapter 3 discusses dynamic performance evaluation methodologies for speculative threads.

- Chapter 4 presents performance tuning policies for speculative threads, as well as runtime support of the execution framework.

- Chapter 5 explores the possibility of building a heterogeneous multi-core system for speculative threads and estimates the upper bound for energy efficiency improvement.

- Chapter 6 deals with the issues in implementing such a heterogeneous system, that is, how to mitigate the overheads and how to allocate resources based on performance characteristics.

- Chapter 7 presents our conclusion and discusses directions for future work.

Chapter 2

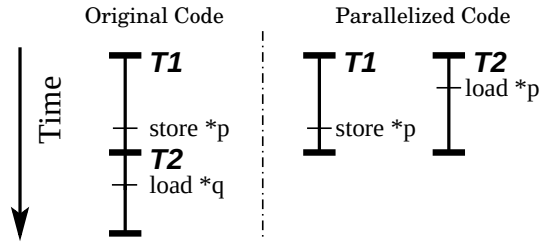
Experiment Infrastructure

In this chapter, we will present the details on the speculative thread execution model as well as our evaluation infrastructure.

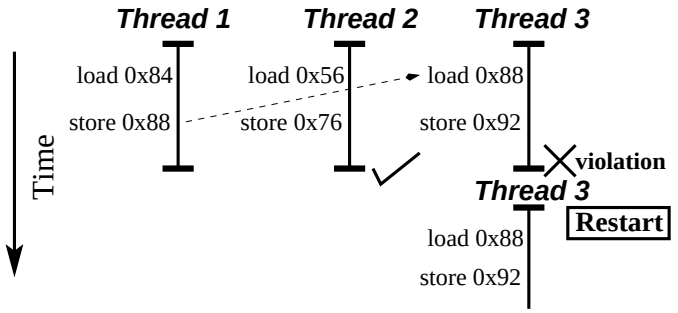
2.1 Speculative Thread Execution Model

The Thread-Level Speculation (TLS) model allows the compiler to parallelize a sequential program without first proving the independence among the extracted threads. During runtime, the underlying hardware keeps track of each memory access, determines whether any data dependence is violated, and re-executes the violating thread(s) as needed. Figure 2.1 contrasts the TLS execution model with the traditional parallelization scheme.

In Figure 2.1(a), the compiler attempts to partition the sequential execution into two parallel threads `T1` and `T2`. As a result, two pointer-based memory accesses, a logically-earlier store and a logically-later load, are allocated to two different threads, and their relative order is inverted in parallel execution. This partitioning is safe only if these two instructions access different memory locations. However, their memory addresses are unknown at the compilation time. Hence the compiler is forced to give up parallelizing this code region. Traditional compilers have very limited capability in



(a) The relative order of memory accesses can be inverted after parallelization. Compiler cannot parallelize this segment of code due to possible memory aliasing.



(b) Code regions with potential dependences are speculatively parallelized. Speculation succeeds in thread 2 and fails in thread 3.

Figure 2.1: TLS can parallelize sequential code regions that are otherwise difficult to parallelize by traditional methods.

parallelizing pointer-based codes.

Figure 2.1(b) illustrates the concept of speculative execution. The threads are numbered according to their original sequential order. If no dependence is violated, the speculative thread commits (`thread 2`); otherwise, it is squashed and re-executed (`thread 3`). TLS empowers compilers to parallelize program regions that were previously non-parallelizable.

2.2 Architectural Support for Speculation

Speculative threads can be supported on a single core with SMT extension (SMT-based) or on multiple cores each running one thread (CMP-based).

We adopted the STAMPede approach to support TLS on a CMP processor [11,

13]. All processing cores have private first-level caches which are connected through a bus to the shared second-level cache. The STAMPede approach extends the cache coherence protocol with two new states, speculatively shared (*SpS*) and speculatively exclusive (*SpE*), and transitions to and from these states. All the speculative threads are assigned with a unique ID that determines the logic order in which the threads commit. The thread ID of the sender piggybacks on all invalidation messages. If a cache line is speculatively loaded, it enters the *SpS* or *SpE* state, and becomes susceptible to dependence violations. If an invalidation message arrives from a logically-earlier thread for that cache line, speculation fails; and the thread is squashed. To recover a speculation failure, all cache lines in the *SpE* state are invalidated and all *SpS* lines transit into the shared state; the thread is then re-executed from the beginning. When a thread is spawned to a busy core that is executing an earlier thread, the new thread is suspended and will be resumed when the earlier thread commits.

The SMT architecture is based on the proposal by Lo *et al.* [45], where processor resources are fully shared by all threads. SMT cores are extensions of superscalar cores of the same issue width, except that renaming unit and register files are replicated. Up to two threads are allowed to fetch instructions in the same cycle based on the ICOUNT fetch policy [46]. Speculative writes are buffered at the first-level cache which is shared by all the SMT threads. No bus transaction is required for inter-thread communications. Each first-level cache line is further extended with bits to indicate whether it is speculatively loaded or modified, and if so, by which thread(s). The rest of the hardware support and implementation details can be found in [14].

2.3 Compilation Infrastructure

Our compiler infrastructure is built on the Open64 Compiler [47], an industrial-strength open-source compiler targeting Intel's Itanium Processor Family. We extended Open64 to extract speculative parallel threads from loops. The compiler estimates the parallel

performance of each loop based on the cost of synchronization and the probability and cost of speculation failure, using loop nesting profile, edge frequency profile, and data dependence frequency profile. The compiler then chooses to parallelize a set of loops that maximize the overall program performance based on such estimations [20]. To dynamically optimize where speculative threads should be spawned, we simply force the compiler to create a different executable in which every loop is parallelized. All binary codes are compiled with `-O3` optimization level. In addition, TLS-specific optimizations, such as inter-thread register and memory resident value communication, and reduction operations, are applied by the compiler to those parallelized loops [48, 49, 50].

2.4 Simulation Infrastructure

We build our simulation infrastructure based on a trace-driven, out-of-order superscalar processor simulator. The trace-generation portion is based on the PIN instrumentation tool [25], and the architectural simulation portion is built on SimpleScalar [51].

The trace generator instruments all instructions to extract information such as instruction address, registers used, memory address for memory instructions, opcode, and etc. The entire trace of the instruction stream is output to disk files.

The simulator reads the trace file and translates the Itanium code bundles generated by the compiler into Alpha-like codes. The simulated pipeline model is based on SimpleScalar [51]. In addition to modeling register renaming, reorder buffer, branch prediction, instruction fetching, branching penalties, and memory hierarchy performance, we also extend the infrastructure to account for different aspects of speculative thread execution, including explicit synchronization through signal/wait, cost of thread commit/squash, and etc. Table 2.1 shows the default architecture parameters of the TLS-enabled Chip Multi-processor (CMP). A set of variation of these parameters will be shown separately when the heterogeneous design space is explored in Chapter 5.

To estimate power consumption of the processors, the simulator integrates the

Table 2.1: Default Parameters of the TLS-enable CMP Processor

Pipeline Stages	Fetch/Issue/Ex/WB/Commit
Fetch/Issue/Commit Width	6 / 4 / 4
ROB/LSQ Size	128 / 64 entries
Integer Units	6 units / 1 cycle
Floating Point Units	4 units / 12 cycles
Private Level 1 Data Cache	64KB, 4-way, 32B
Private Level 1 Instruction Cache	64KB, 4-way, 32B
Memory Ports	2 Read, 1 Write
Branch Predictor Level1/Level2 Size History Register	2-Level Predictor 1 / 1024 entries 8 bits
Branch Mis-prediction Latency	6 cycles
Number of Cores	4
Shared Level 2 Unified Cache	2MB, 8-way, 64B
L1/L2/Memory Access Latencies	1 / 18 / 150 cycles
Thread Squash/Spawn/Sync Latencies	5 / 5 / 1 cycles

Wattch [52] model for core power consumption, the Cacti [53] model for cache power consumption, and the Orion [54] model for interconnection power consumption. We assume the 70nm technology.

2.5 Benchmark Workloads

Our target workloads are benchmark programs from the SPEC CPU2006 and CPU2000 suites written purely in C and C++. If a benchmark program appears in both suites, only one instance is evaluated. For example, we have evaluated 401.BZIP2 from SPEC CPU2006 but omitted 256.BZIP2 from SPEC CPU2000. For benchmark 175.VPR, we treated the PLACE and ROUTE input sets as two separate cases due to their distinct differences. They are referred as VPR-P and VPR-R in later chapters. All benchmarks are evaluated using the *ref* input sets. When there are multiple input sets, the first one is chosen for evaluation.

The following benchmark programs are not evaluated due to technical difficulties:

252.EON, 447.DEALII, 471.OMNETPP, 483.XALANCBMK and 450.SOPLEX. In addition, 444.NAMD and 453.POVARY are not evaluated in Chapter 3 and 4, and 164.GZIP and 255.VORTEX are not evaluated in Chapter 5 and 6. Nevertheless, we have evaluated our proposals on 25 different benchmark programs that exhibit diverse characteristics.

To reduce simulation time, we have adopted the SimPoint-based sampling technique [55] with 100 million instructions per sample and up to 10 samples per benchmark. Up to one billion instructions can be simulated for each benchmark. With the sample size of 100 million instructions, the side effect of warming-up is negligible.

Chapter 3

Performance Estimation

To extract speculative threads, the compiler or the runtime system must first perform trade-off analysis to determine the performance impact of parallelization. Existing work mostly relied on compilers to statically analyze the program, and then extract speculative threads. We refer to those as *static* thread management. In this chapter, we first explain the limitation of static approaches and propose and evaluate our *dynamic* performance prediction scheme.

3.1 Limitations of Static Approaches

In static approaches, compilers often analyze extensive profile information to estimate the performance impact of speculative threads and then determine where and how to create speculative threads. Being able to perform global and even inter-procedure analysis, compilers can extract coarse-grain parallelism in which speculative threads may contain several thousand instructions. However, it is difficult and sometimes impossible for compilers to accurately estimate the performance impact even when extensive profile information is available. We have identified four key reasons.

3.1.1 Profiling information is insufficient for estimating the costs of speculation, synchronization and other overheads

The performance impact of speculation is determined by the number of useful execution cycles in the speculative threads that can overlap with the execution of the non-speculative thread. To determine this overlap, the compiler must determine the size of consecutive speculative threads, the cost of speculation failures and the timing of their occurrence, the cost of synchronization, and the cost of managing speculation.

However, these factors are often difficult to estimate even with accurate profiling information. For example, the rate of speculation failures not only depends on the number of inter-thread data dependences, but also on the timing of its occurrence. For loops with complex control flow, it is difficult to determine whether the load/store instructions in consecutive iterations are dependent; and if they are, which loads will cause dependence violations at runtime. Probability-based data and control dependence profiling, which is used in many TLS compilers, is insufficient to come up with such estimations.

3.1.2 Performance impact of speculative threads depends on the underlying hardware configuration

Because speculative threads must share the underlying hardware resources, the configuration of the underlying hardware can change the behaviors of these threads. In particular, interaction between the speculative threads and the cache components has a profound impact on performance. On the one hand, speculative threads, even when they fail, can potentially bring data items into the cache and improve the performance of the non-speculative thread. On the other hand, speculative threads can modify data items that are shared with the non-speculative thread and introduce misses that otherwise do not exist. Furthermore, the CMP architecture effectively increases total cache size. However, as data is spread across multiple caches, CMP introduces coherent misses that

do not exist in sequential execution. The impact of such cache behaviors is difficult for the compiler to determine even with accurate profile information.

3.1.3 Speculative thread behaviors are input-dependent

The performance of speculative threads is often dependent on the characteristics of the input data. TLS takes advantage of probabilistic data dependences by speculatively assuming that these dependences do not exist. This mechanism is beneficial only if these data dependences are infrequent. Frequently occurring data dependences should be synchronized. Choosing the threshold that separates frequent and infrequent dependences is a delicate matter, since a high threshold leads to excessive speculation failures, and a low threshold leads to serialization of the threads. However, once this threshold is chosen and the set of frequently occurring dependences are synchronized, this decision is compiled into the binary, even if the decision is not proper for some input sets.

When extracting speculative threads for 256.BZIP2, we collected profiles using the `train` input set and decided which dependences to speculate on. When the program executes with the three `ref` input sets `source`, `graph`, and `program`, we found that the percentage of total execution cycles that are wasted due to speculation failure was 25%, 40%, and 31%, respectively.

Therefore, speculative threads that are created to improve performance under one workload may potentially degrade performance when the input set changes.

3.1.4 Speculative threads experience phase behavior

For some applications, it has been reported that the same codes may exhibit different performance characteristics as the program enters different phases of execution [56]. We refer to this behavior as phase behavior. In the context of TLS, phase behavior can be manifested as changing the effectiveness of speculative threads — speculative threads that improve performance during certain phases of execution can potentially degrade performance during other phases of execution. Probabilistic profiles cannot capture this

behavior, as speculative decisions that are compiled statically into the binary cannot adapt to this behavior.

Phase change is a natural phenomenon in real-world applications, and can occur as a result of ordinary programming constructs. For example, in algorithms that search for the maximum or minimum in a large data set, the frequency of updating the global variables decreases as the algorithm progresses. Thus a loop that is not fit for speculative execution earlier in the program can become a good candidate during later phases of the execution.

In summary, if the execution of speculative threads can be monitored, it is possible for the runtime system to determine the impact of the above factors. Therefore, dynamically managing speculative threads can be an attractive alternative.

3.2 Determining the Performance Impact

In this section, we propose a way to quantitatively determine the efficiency of speculative thread execution at runtime. Our technique builds on execution cycle breakdowns that can be obtained through hardware-based, programmable performance monitors. Details of these monitors and how to obtain the breakdowns will be addressed in Chapter 4.1.

For our purposes, cycles for TLS execution are broken into six segments:

- **Busy**: cycles spent graduating non-TLS instructions;
- **ExeStall**: cycles stalled due to lack of instruction-level parallelism;
- **iFetch**: cycles stalled due to instruction fetch penalty;
- **dCache**: cycles stalled due to data cache misses;
- **Squash**: cycles wasted due to speculation failures; and
- **Others**: cycles spent on various TLS overheads, including thread spawning, committing, and synchronization; as well as idling cycles due to unbalanced workloads.

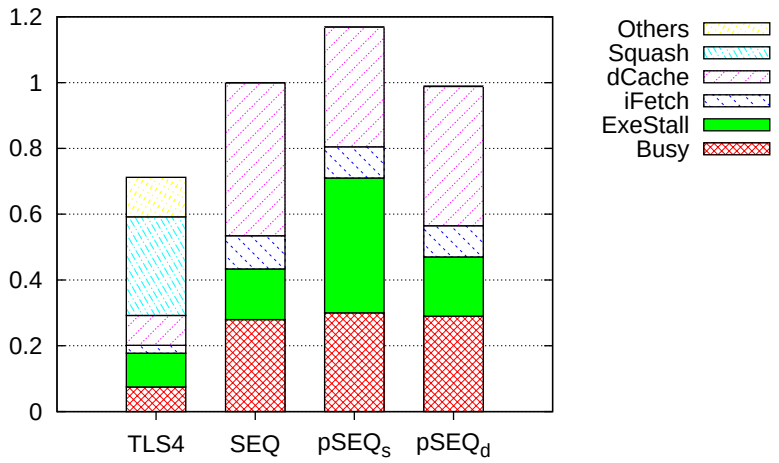


Figure 3.1: Sample execution time breakdown of a loop invocation under two execution models: TLS on four core (TLS4) and sequential execution (SEQ). Sequential execution time of the loop predicted from the cycle breakdown of TLS4: a simple prediction $pSEQ_s$ and a detailed prediction $pSEQ_d$. Bars are normalized to SEQ.

Figure 3.1 shows the execution time breakdown, normalized to the execution time of the sequential execution, of a loop executing in TLS mode on four cores (**TLS4**) and executing sequentially (**SEQ**). Each segment in `TLS4` is the aggregated cycles scaled down by *four* to show the relative speedup compared to `SEQ`.

To isolate the performance impact of the speculative threads, we attempt to *predict* the sequential execution time from the execution time breakdown of the TLS execution. A straightforward prediction, shown as bar $pSEQ_s$ in Figure 3.1, is to subtract the `Squash` and `Others` cycles from the total aggregated cycles of parallel execution. Each segment in $pSEQ_s$ is just four times of its counterpart from `TLS4`, since the later is scaled down by four. In this prediction, `Busy` can be accurately predicted because the amount of useful work done in TLS mode and sequential execution is similar; `iFetch` is also similar in these two execution modes. However, execution stall and cache behaviors can change dramatically when the sequential program is decomposed into multiple threads. To improve our prediction, a more accurate model is developed to address the inaccuracies in predicting `ExeStall` and `dCache` segments.

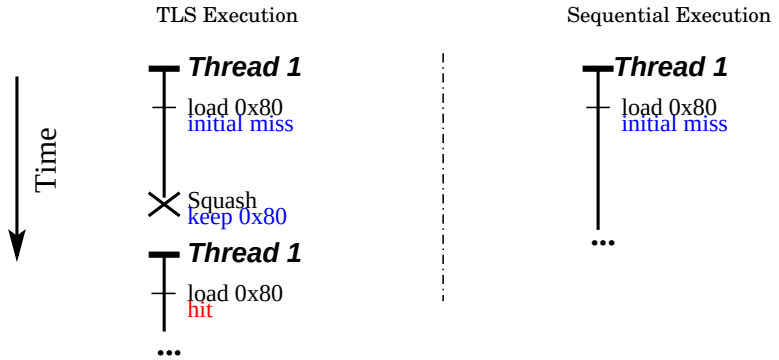
3.2.1 Adjustment to execution stall

When the original program executes sequentially on one core, instructions from multiple iterations of the same loop are available for scheduling, and thus the core is able to effectively exploit instruction-level parallelism (ILP). However, when the same code is decomposed into multiple threads that are distributed to multiple cores, execution stall may increase since fewer instructions are available for scheduling. This effect correlates with the average number of dynamic instructions per thread (defined as *thread size*). When the thread size is much greater than the reorder buffer (ROB) size, the variances of execution stall between sequential and TLS execution can be negligible. However, when the thread size is smaller than or comparable to the ROB size, execution stall will increase considerably. Thus, to accurately predict execution stall in sequential runs, the execution stalls in parallel runs must be scaled down by a factor that is correlated to the ROB size of the processor and inversely correlated to the thread size. In our experiments, the following estimation is demonstrated to be effective for approximating this relationship:

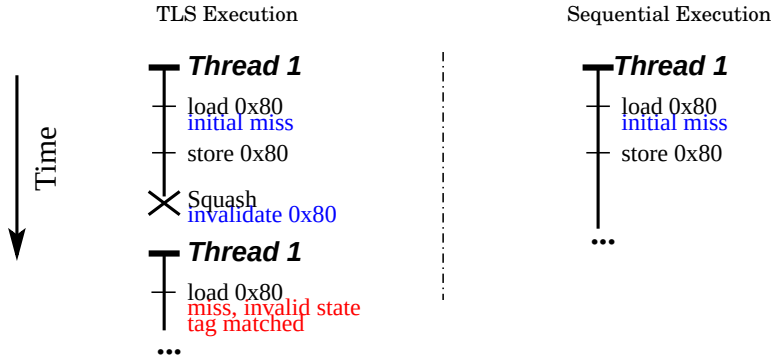
$$1 + \frac{ROB\_Size}{Issue\_Width * Thread\_Size} \tag{3.1}$$

3.2.2 Adjustment to data cache behaviors

Speculative threads have a significant impact on the first-level data cache (L1D) performance. Because speculative states are buffered at L1D, and L1D cache misses observed in speculative parallel execution may or may not occur during sequential execution; and vice versa. On the other hand, the second level unified cache (L2) is responsible for caching the working set for the entire process. Since the aggregation of all committed instructions on all cores in speculative parallel mode is the same as in the sequential mode, the two modes generate similar access patterns to L2. Thus, the performance of L2 remains mostly unchanged and we do not consider the performance impact of speculation on L2.



(a) In TLS execution, a data item loaded into the L1D cache by a speculative thread that is eventually squashed is re-used when the thread re-executes with no additional cache miss. In sequential execution, this access incurs one initial cache miss.



(b) In TLS execution, when data modified by a squashed speculative thread is re-used, this access incurs an additional cache miss. In sequential execution, there will be no additional cache miss.

Figure 3.2: Impact of failed speculation on L1 data caches

Our main approach is to classify L1D cache misses that occur during speculative parallel execution, and predict whether they would occur during sequential execution. We have identified the following scenarios that require special handling: (i) data brought into the L1D cache by a speculative thread that is eventually squashed; (ii) data modified by a speculative thread that is eventually squashed; (iii) data brought into a neighboring L1D cache; and (iv) data evicted from one L1D cache but available in other L1D cache(s). In the rest of this section, we will describe the four scenarios in detail.

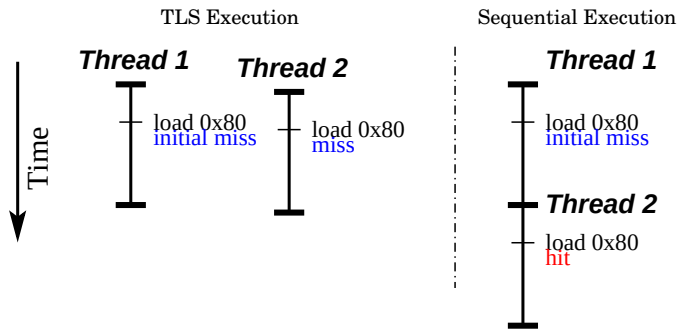
Scenario I: Speculative threads that bring data into L1D can be squashed and then

re-executed. When this occurs, a straightforward classifier would count all the cycles, from the point of thread starting to the point of squashing toward `Squash`. Note that this classifier counts all stalls due to L1D cache misses as part of `Squash`. However, this simple classification can be inaccurate if the data is used when the thread re-executes. This is because when the squashed thread re-executes and accesses the data for a second time, it will not incur another cache miss, as shown in Figure 3.2(a). However, since `Squash` cycles are discarded in the prediction, the latency associated with this memory access is inadvertently not counted towards the sequential execution time. To rectify this counting inaccuracy, cycles stalled due to L1D miss during the squashed execution must be recorded separately and nevertheless contribute to the predicted sequential execution time.

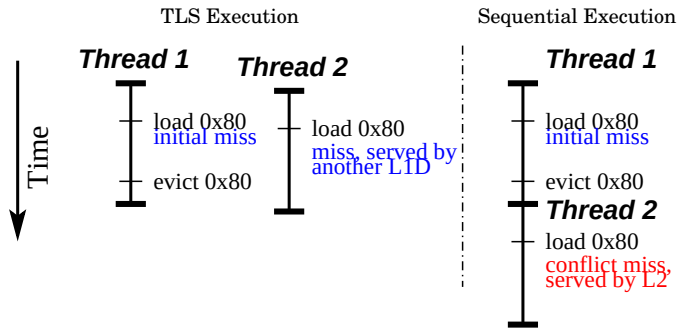
Scenario II: When a data item is modified by the speculative thread that is eventually squashed, the cycles must be counted differently. As shown in Figure 3.2(b), the squashing mechanism invalidates the modified copy in the cache to ensure correctness. During re-execution, this data item will be re-loaded into the L1D cache, and thus incur another cache miss. In sequential execution, the second cache miss would not occur. This case can be identified in speculative parallel execution, since the second miss would find an invalidated cache line with a matching tag.

Scenario III: Multiple speculative threads concurrently executing on multiple cores can further complicates L1D cache performance. When two threads on different cores load from the same address, they incur two L1D cache misses, as shown in Figure 3.3(a). However, in sequential execution, it is likely that the logically-later miss on the second thread becomes a hit since the threads are executing on the same core. Thus, to correctly predict sequential performance, latency associated with L1D caches misses that return the data item in shared state must be omitted from the prediction.

Scenario IV: Speculative threads run on multiple cores, and thus are able to utilize multiple L1D caches. This can increase the effective associativity of the first level cache



(a) Two threads load from the same address on separate cores incur two separate cache misses, and the cache line results in the shared state. In sequential execution, these two misses are likely to incurs only one cache miss.



(b) A conflict miss served by lower level cache may be served by another L1D cache in parallel execution. This leads to a lower cache access latency.

Figure 3.3: Impact of speculative parallel execution on L1 data caches

compared to the sequential execution. Thus, conflict misses that occur in the sequential execution may or may not occur in the parallel execution. As shown in Figure 3.3(b), the conflict miss served by the second level cache in the sequential execution does not occur in parallel execution; rather, it is served by a neighboring L1D cache. This results in a different latency compared to accessing L2. Therefore, we need to compensate the predicted sequential execution time with an L2 access latency each time this scenario occurs. To count the number of such occurrences, we extend each cache line with one additional bit to track whether this cache line is loaded by a logically-later thread. We refer to this bit as *ExtraShared*. If a cache line with the *ExtraShared* bit set is

evicted, it is *likely* that the load in a logically-later thread would result in a conflict miss in sequential execution, so the count is incremented by one. Details regarding this extension will be discussed in Chapter 4.1.2. This method is simple and with minimal hardware support. It can capture most of the conflict misses, but not all situations. For instance, if the eviction happens in thread 2 (Figure 3.3(b)), it will not be counted because the *ExtraShared* bit is associated with the cache on which the first thread executes. Moreover, if a cache line loaded by more than one logically-later thread is evicted, this method will only increase the count by one, in which case multiple increments are desired.

It is worth pointing out that cache behaviors make it difficult for static analysis to derive the impact of speculative threads accurately. If speculation failure often helps to fetch useful data into the L1 cache, a high failure rate can be benign, but if failed threads often invalidate useful data, even a moderate failure rate can be detrimental. In Chapter 4.1.2, we will provide a detailed description of the hardware performance monitors needed to classify the misses described above.

3.3 Prediction Accuracy Evaluation

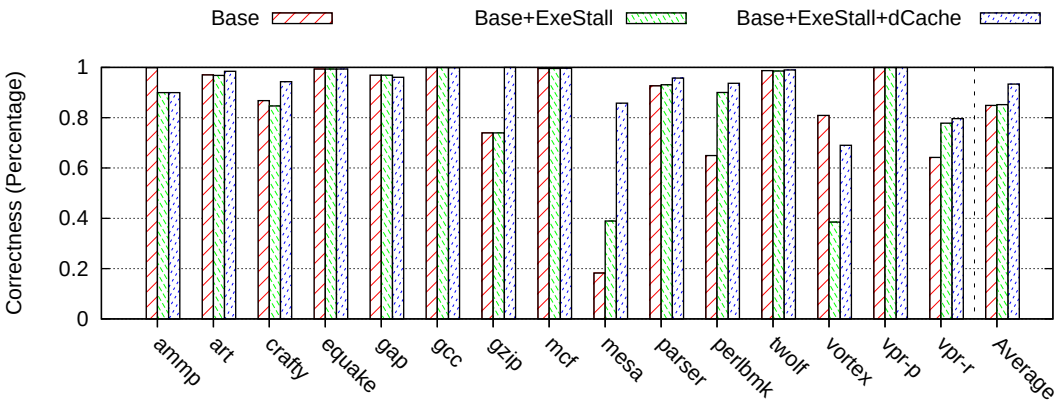
In this section, we evaluate how accurate we are able to predict sequential performance from speculative parallel execution. To simplify our evaluation, we focus only on loops that are selected for parallelization by the compiler [20]. These loops are normally important and representative. For each loop invocation, three sets of execution cycles are obtained: the sequential execution cycle T_{SEQ}, the TLS execution cycle T_{TLS}, and the sequential cycle predicted from parallel execution time breakdown T_{pSEQ}. We use two metrics for evaluation: (i) the percentage of total loop execution we are able to correctly determine whether or not parallel execution outperforms sequential execution (*correctness measure*); and (ii) the difference, in cycles, between the predicted sequential execution and the real sequential execution cycle (*similarity measure*). We evaluated

both metrics on three increasingly complex schemes: `Base` corresponds to the baseline prediction scheme described in Chapter 3.2; `Base+ExeStall` incorporates the execution stall scaling described in Chapter 3.2.1; and `Base+ExeStall+dCache` further incorporates the data cache behavior classification described in Chapter 3.2.2. Performance evaluation on how these three schemes work with the rest of the system can be found in Chapter 4.3.2.

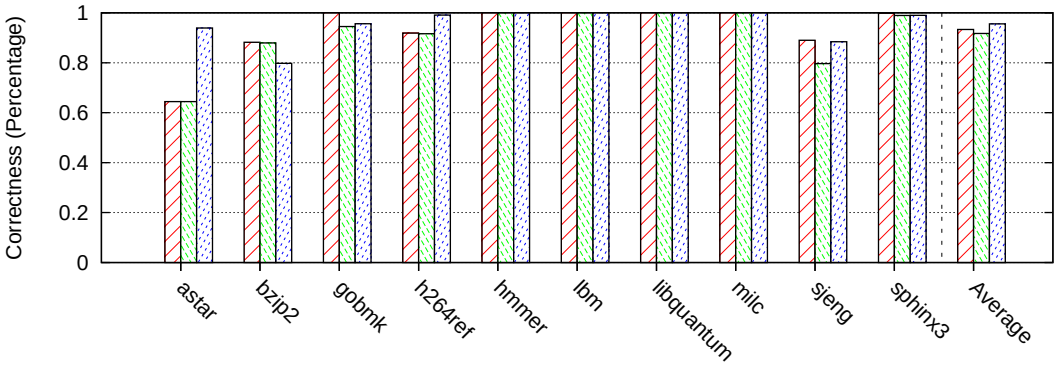
3.3.1 Evaluation using the correctness measure

For a loop invocation, if T_{pSEQ} and T_{SEQ} are both greater or both smaller than T_{TLS}, the prediction is considered correct, otherwise it is incorrect. We categorize loops into three groups: (i) loops whose parallel and sequential performance differ by less than 5%, the rest of the loops that are (ii) correctly predicted, and (iii) incorrectly predicted. The first loop category is excluded in the correctness measure since whether or not these loops are parallelized contributes little to the final performance.

Figure 3.4 shows the degree of correctness. Overall, the correctness measure improves as the proposed execution stall scaling and cache miss cycle classification are incorporated. With execution stall scaling, the correctness is significantly improved in MESA, PERLBMK and VPR-R. With cache miss cycle classification, the correctness is further improved, especially in CRAFTY, GZIP, MESA, ASTAR, and H264REF. There are three exceptions, AMMP, VORTEX and BZIP2, however, that do not follow this trend of improvement, and `Base+ExeStall+dCache` yields noticeably lower correctness than `Base`. The reason is that `Base` in these benchmarks heavily biases towards one side of prediction that *happens* to match the actual situation. The extensions to the prediction remove such biases but result in lower degree of correctness. Nevertheless, `Base+ExeStall+dCache` always yields the highest similarity in these three benchmarks (next subsection), showing its effectiveness. Overall, `Base+ExeStall+dCache` correctly predicts 94.3% of the loops, a 6.8% improvement compared to the `Base` scheme.



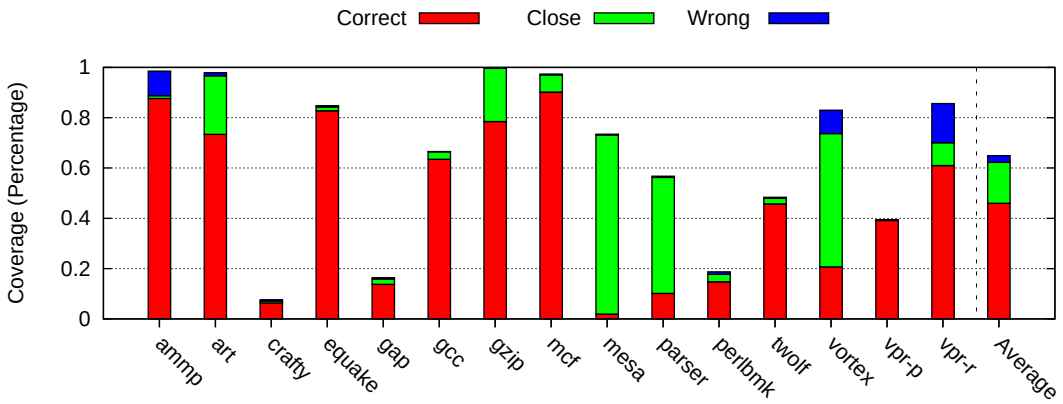
(a) SPEC CPU2000 Benchmarks



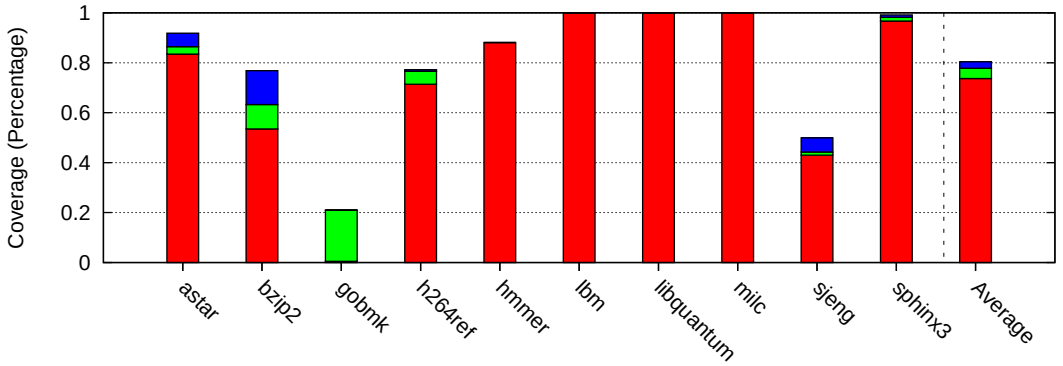
(b) SPEC CPU2006 Benchmarks

Figure 3.4: Correctness measure: dynamic coverage of parallelized loops that are *correctly* predicted for different prediction techniques.

Figure 3.5 shows the dynamic coverage in execution time of the three loop categories: `Close` represents loops whose performance differs by less than 5% after being parallelized; `Correct` and `Wrong` stand for the rest of the loops that is correctly and incorrectly predicted by the last scheme `Base+ExeStall+dCache`, respectively. Note that the sum of the three categories show the percentage of speculative parallel execution in each benchmark. Our prediction scheme generally works well: on average, only 2.6% of total execution time suffers incorrect prediction.



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

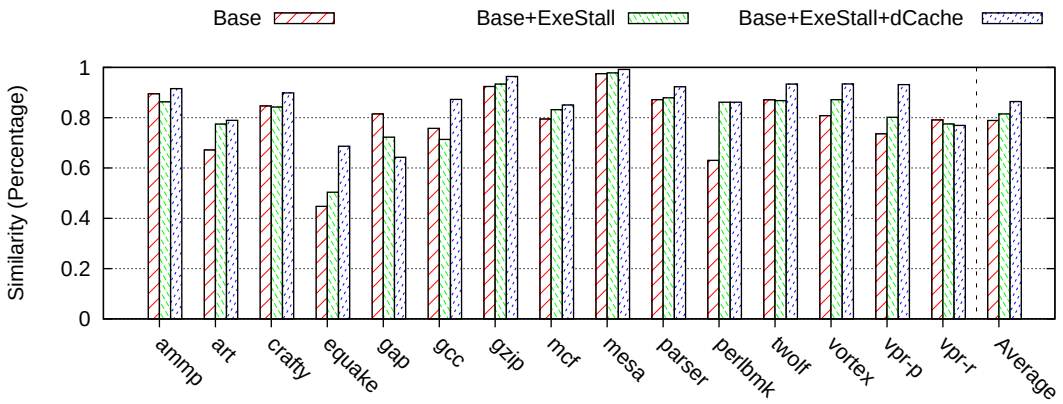
Figure 3.5: Dynamic coverage: percentage of execution time spent on parallelized loops over the entire benchmark.

3.3.2 Evaluation using the similarity measure

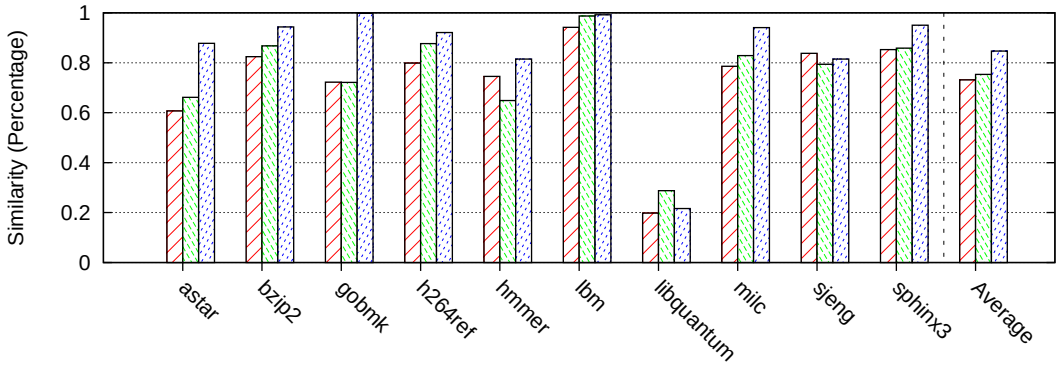
The similarity measure quantitatively evaluates the accuracy of our prediction for *all* loops parallelized. We first define *dissimilarity* as the accumulative difference between T_{pSEQ} and T_{SEQ} over the sum of T_{SEQ} for all loop invocations, i.e.,

$$dissimilarity = \frac{\sum |T_{pSEQ} - T_{SEQ}|}{\sum T_{SEQ}} \tag{3.2}$$

Similarity is defined as $1 - dissimilarity$. The results are shown in Figure 3.6. Generally, similarity improves with the incorporation of execution stall scaling and data cache miss



(a) SPEC CPU2000 Benchmarks

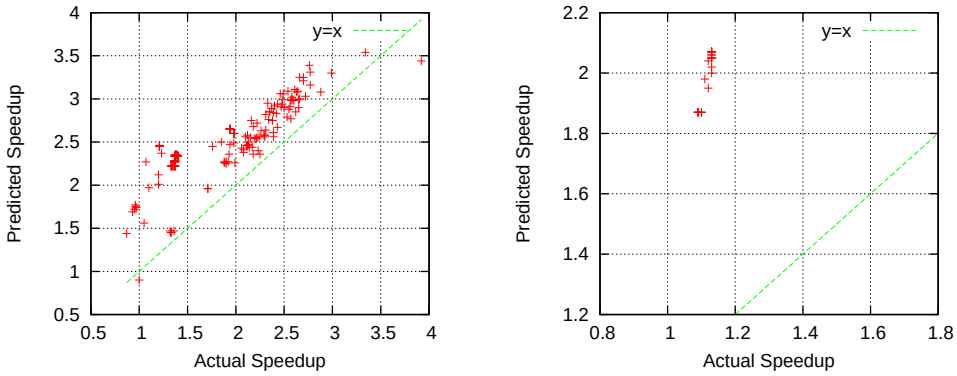


(b) SPEC CPU2006 Benchmarks

Figure 3.6: Similarity measure: how close our prediction could approximate the actual sequential execution cycle.

classification. Overall, the improvement is gradual and significant: 76.6% for `Base`, 79.0% for `Base+ExeStall`, and 85.7% for `Base+ExeStall+dCache`.

There are, however, a few exceptions: GAP, VPR-R, LIBQUANTUM and SJENG. For these benchmarks, the predicted stall cycles due to cache misses are less than that of the actual sequential execution. This is because there are less conflict misses in the parallel execution, due to the increase in effective associativity when threads are spread across multiple cores. Although we attempt to compensate for this effect, our mechanism does not cover all possible situations, as stated in Chapter 3.2.2.



(a) Benchmark EQUAKE (b) Benchmark LIBQUANTUM

Figure 3.7: Comparing the predicted speedup and the actual speedup for individual loop invocations.

We tap into two of these benchmarks, EQUAKE and LIBQUANTUM, to find evidence. Figure 3.7 is a 2D scatter graph, where each data point corresponds a loop invocation. The y- and x-axis are the predicted speedup and the actual speedup, respectively. In other words, the y- and x-coordinate of a point represent T_{pSEQ}/T_{TLS} and T_{SEQ}/T_{TLS}. Due to the massive number of loop invocations, each point in EQUAKE is an accumulation of 3000 consecutive invocations of the same loop. The dotted line $y = x$ corresponds for perfect prediction.

In EQUAKE, almost all points fall above the dotted line, but form a linear relationship with the same slope as the dotted line. This indicates that our prediction is *systematically* optimistic, i.e., predicting less cycles for sequential execution, due to the same reason of not compensating for all conflict misses. Similar situation is found in LIBQUANTUM as well. We have also observed that the L1 cache replacement rate is very high in both EQUAKE and LIBQUANTUM. This is a strong indication that parallel execution could have reduced conflict misses compared to sequential execution. Also note that these benchmarks achieves near perfect prediction in terms of correctness measure, thus similarity measure is only used as tie breakers. Therefore, as long as the prediction is consistently pessimistic or optimistic across all loops, the dynamic tuning system is

able to make proper decisions.

3.4 Related Work

Most prior works on deciding where to extract speculative threads use static analysis [6, 15, 16, 17, 18, 19, 20, 21]. These works will be discussed in Chapter 4.5. There are only a few that have some dynamic features [23, 24, 22, 20, 57, 58].

Marcuello *et al.* [24] and Tubella *et al.* [23] propose hardware to dynamically detect loops and gather loop performance characteristics and then use this information to speculatively parallelize programs. However, it is difficult for the hardware to identify high-level program structures. Therefore, hardware-based mechanisms tend to parallelize inner loops. In our approach, all loops are considered as potential candidates for parallelization and later incorporated as compiler annotation (Chapter 4) to prevent the runtime system from only parallelizing inner loops.

Johnson *et al.* [22] and Wang *et al.* [20] propose using a compiler to instrument a profile run to search through candidate loops. The key difference between these thread partition techniques and our proposal is that the former techniques search for speculative threads using profile information through sample runs with some training input sets and then compiles the choices into the binary, whereas the latter does the search at runtime. Thus, only our proposal can adapt to performance characteristics variations associated with different input sets and phase changes.

The works from Renau *et al.* [57, 58] are the most closely related work. Frequency of squashes and resource availability are used to determine whether to prevent a thread from re-spawning. This simple metric is effective in preventing the system from useless re-spawning. In this thesis, our approach requires the capability for determining whether speculation improves performance compared to sequential execution, and more importantly, by how much. Therefore, more detailed runtime information that incorporates cache performance is collected and a more sophisticated methodology is proposed

to *quantitatively* evaluate the performance impact of speculative threads.

3.5 Summary

In this chapter, we have explained the limitations of traditional static thread management schemes. To overcome these limitations, we propose to dynamically evaluate the performance impact of speculative threads. Our approach is to predict the sequential execution time and deem the difference between the actual parallel execution and the predicted sequential execution as a metric that quantitatively evaluates the efficiency of speculative threads.

Two important adjustments, execution stall scaling and cache behavior classification, are proposed to make the prediction more accurate. In the final scheme, we are able to make correct predictions for 94% of the loops and approximate the actual execution time by 86%.

To summarize, we believe that our performance prediction scheme is accurate enough for the runtime system to make parallelization decisions. We also believe that being able to accurately understand cache behaviors and the effects on execution stall is important for determining the performance impact of speculative threads.

Chapter 4

Dynamic Performance Tuning

In the previous chapter, we have discussed a proposal to quantitatively evaluate the efficiency of speculative threads as they execute on the fly. Another important issue for the dynamic tuning system is how to select where in the sequential program to spawn those threads based on their efficiency. This decision has to be judicious because it is possible to have multiple choices of spawning points that overlap with one another.

In this chapter, we first present our execution framework, and then propose a spectrum of tuning policies that aim to search for the best set of spawning points. We will evaluate the performance of these tuning policies as well as the entire system and compare it with the state-of-the-art static approach. We will also showcase a few benchmarks to give insights on why dynamic tuning works better than static analysis.

4.1 Dynamic Execution Framework

To support dynamic performance tuning, the execution framework needs to be able to adjust the behaviors of speculative threads. This is done through runtime thread management. At a fine granularity, the framework needs to program hardware performance counters to collect necessary information and store the tuning decisions on where to spawn speculative threads.

4.1.1 Runtime thread management

Monitoring: The performance profile of speculative threads must be collected dynamically. Such profiles can be application-dependent, such as loop iteration count; architecture-dependent, such as memory access latency; or both, such as cache miss rate. Hardware-based performance counters [59, 60] are programmed to collect such information (Chapter 4.1.2). A small piece of code that initializes these counters is executed at the beginning of execution by modifying a `libc` entry-point routine named `__libc_start_main`. This mechanism has been proposed by Lu *et al.* [30].

Evaluation: Once the profile is collected, it is analyzed to determine the efficiency of speculative threads, using the methodologies presented in Chapter 3. Based on the analysis result, the dynamic tuning policy then searches for the optimal spawning points. The analysis and search results are stored into a hardware-based decision table that is kept in sync with all processor cores (Chapter 4.1.3).

Adjustment: At the beginning of each candidate spawning point, the runtime system queries the decision table and decides whether a speculative thread should be created. If no entry is found for a particular candidate, it will be given a chance to spawn threads and be evaluated. The adjustment of thread behaviors is a direct result of the tuning policy (Chapter 4.2) as well as the performance prediction scheme (Chapter 3.2). Their impacts on system performance will be evaluated in Chapter 4.3.2.

4.1.2 Programming hardware performance counters

Our performance prediction relies on cycle breakdowns of speculative parallel execution. Note that the goal is not to build accurate TLS execution cycle breakdowns, but to generate a set of components to help reconstruct the sequential execution cycle.

Obtaining cycle breakdowns in an out-of-order processor is difficult due to the overlap of multiple on-the-fly instructions. Examining the instructions at the head of reorder

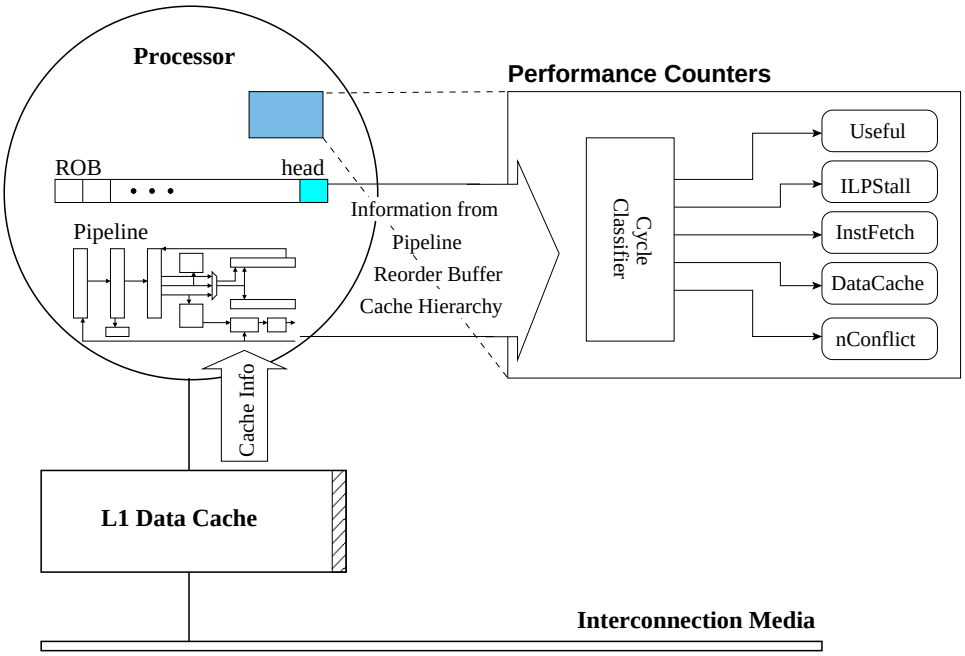


Figure 4.1: Each core is augmented with hardware performance counters to classify cycle stalls.

buffer (ROB) gives us some clues [60] to the causes for stalls. We propose a cycle classification scheme that works on programmable hardware counters that is common on modern processor chips. Figure 4.1 depicts our use case that aims to obtain the following breakdown components:

- `Useful`: cycles spent graduating useful instructions, i.e., instructions not related to speculative thread management;
- `ILPStall`: cycles stalled due to lack of instruction-level parallelism;
- `InstFetch`: cycles stalled due to instruction fetch penalty;
- `DataCache`: cycles stalled due to data cache misses; and
- `nConflict`: the number of conflict misses that need to be compensated when predicting sequential execution time.

Note that all TLS-related overheads, including thread idling and squashing, are not attributed to any counter because they would not be present in sequential execution.

Basic cycle classification

The cycle classification scheme is explained below. As a simplification, we describe cycle classification for a processor that commits one instruction per cycle. For processors with higher commit width, multiple counters can be incremented in a single cycle, based on the activity of each graduation slot. At *each* cycle, the classifier examines the head of the ROB. If the ROB is empty, the `InstFetch` counter is incremented by one. At the head of the ROB, if the instruction is unable to graduate, a hardware *accumulator* is incremented by one to keep track of the number of cycles this instruction stalls. When the instruction finally commits, no counter is incremented if this instruction is related to TLS thread management. Otherwise, the `Useful` counter is incremented by one, and the value in the accumulator is added to the `DataCache` counter if this is a memory instruction and results in a cache miss, or added to the `ILPStall` counter otherwise. The accumulator is reset whenever an instruction graduates.

This mechanism is similar to the performance monitors in IBM POWER5 [60]. However, the following situations require special handling for TLS execution: (i) aggregating counters across multiple cores; (ii) re-distributing counters when a thread is squashed due to speculation failure; and (iii) categorizing cache misses.

Aggregating counters across multiple cores

Speculative parallel threads are distributed across multiple cores, and thus performance counters must be aggregated to obtain the complete cycle breakdown. There is only one non-speculative thread and only the non-speculative thread is allowed to commit. At thread commit time, the non-speculative thread forwards its performance counters to its successor, and makes the successor non-speculative. The new non-speculative thread adds the forwarded values to its own corresponding counters. Therefore, when a

speculatively parallelized region completes, the counters on the core that commits the last thread contain the complete breakdown.

Handling thread spawning and squashing

When a speculative thread is spawned to a processing core, all counters on that core are reset. When a speculative thread is squashed, all counters except `DataCache` and `nConflict` on that core are reset. Note that we do not attempt to record the cycles wasted due to squashing, but need to preserve these two counters for the reasons explained in Figure 3.2(a).

Cache miss categorization

When a load instruction that stalls due to a cache miss finally commits, Chapter 3.2.2 describes four scenarios each requiring a different classification mechanism based on information returned from the cache.

Upon a first level data (L1D) cache miss, we must check whether the cache access has found an invalidated cache line with a matching tag. If so, it indicates that this cache line has been loaded into L1D, but then speculatively modified by a thread that is eventually squashed. At this point, the original data must be re-loaded from the next level cache. This cache miss would not have occurred if this program is executed sequentially (Figure 3.2(b)). Thus, the stall cycles in the accumulator are discarded.

If a cache miss is served by the L1D cache of a different core (feedback from underlying cache coherence protocol), it is possible that this cache miss would not have occurred in sequential execution, as illustrated in Figure 3.3(a). This cache miss behaves like a coherence miss. Since coherence misses cannot occur during sequential execution, the stall cycles in the accumulator is also discarded.

Figure 3.3(b) depicts a different scenario, where a cache line brought in by an earlier load is evicted due to replacement. Re-referencing to this cache line will cause a cache miss, and it is possible for this cache miss to be served by a neighboring L1D cache. In

this case, the same coherence messages will be exchanged between the caches as before; however, this cache miss is likely to incur a conflict miss in sequential execution. In other words, conflict misses in sequential execution can behave similarly to coherence misses in TLS execution. However, stall cycles that correspond to coherent misses are always discarded for predicting sequential execution time. Thus, a new mechanism must be introduced to compensate this effect. We propose to augment each L1D cache line with one bit and introduce a new counter to count the number of occurrences of such cache misses in each core. We refer this bit as *ExtraShared* and the counter as `nConflict`. The scheme works as follows: (i) *ExtraShared* is initially set 0 for all cache lines when TLS execution starts; (ii) when a cache line is fetched to serve a cache miss from another cache, set the *ExtraShared* bit for that cache line; (iii) when a cache line is replaced, increment the `nConflict` counter if the *ExtraShared* bit is set, and reset *ExtraShared* bit; (iv) when the non-speculative thread commits, the value of `nConflict` is forwarded and accumulated as other cycle counters.

Essentially, the `nConflict` counter predicts how many additional conflict misses might have occurred in sequential execution. During performance evaluation, we compensate the predicted sequential execution cycle with the value in `nConflict` multiplying the second level cache access latency.

Note that this scheme cannot catch all conflict misses that occur in sequential execution but not in speculative parallel execution, as observed in Chapter 3.3.2. More sophisticated approaches can potentially alleviate this situation, such as replacing the *ExtraShared* bit with a counter in each cache line. However, our simple scheme can provide accurate prediction in most situations, thus we did not pursuit alternatives with higher hardware costs.

4.1.3 Maintaining the decision table

The hardware-based decision table is built in each processor core. It is a content-addressable memory (CAM) indexed by a unique identification number associated with

each candidate spawning point, namely its instruction address. Each table entry contains two fields: a *saturation counter*, which is incremented if the TLS execution outperforms the predicted sequential execution and decremented otherwise, and a *performance summary*, which contains the cumulative difference in execution time (i.e., cycles) between the TLS execution and the predicted sequential execution. Note that the performance summary can be a negative value.

Before a candidate spawning point is executed, this table is consulted for whether to grant or decline this spawning request. A decline would fail the spawning instruction and serialize the requesting candidate, given precedence to other overlapping candidate threads. When speculative threads complete execution, the processor that commits the last thread has the aggregated information from all other processors and updates the corresponding table entry: (i) increase or decrease the saturation counter, and (ii) add the difference between the TLS execution and the predicted sequential execution cycle to the performance summary. The table update is then broadcast to other processors. This operation is infrequent; it only happens when the entire invocation of speculative threads finishes.

Essentially, the two fields in a table entry store both the qualitative and the quantitative evaluations for each candidate spawning point. They are used by the dynamic tuning policies described next.

4.2 Performance Tuning Policies

Our TLS system focuses on parallelizing loops from sequential programs. Many such programs contain multiple nested loops, and thus the dynamic tuning policy is required not only to identify and parallelize loops that can benefit from TLS, but also to select the right level of loop to maximize the overall performance gain. A straightforward mechanism is to first tentatively parallelize each loop, measure the performance impact, and then serialize the ones for which TLS execution is ineffective. However, there can

be various ways to determine the order in which loops in a loop nest are evaluated and to decide the precedence among different loop levels. In this section, we first examine the design issues for the dynamic tuning system and then build the most effective policy in incremental steps.

4.2.1 Basic design issues

An effective policy should first identify loops that lead to overall speedup. A loop can have different execution times in different invocations across the program. One example is a loop traversing a linked list: the execution time of the loop is dependent on the length of the linked list, which may vary from one list to another. Another example is a loop with conditional break-out statements. For such loops, performance estimation based on one invocation could be misleading. Consider a loop with 6 invocations, for example; the first one takes a long time to execute and TLS is 2000 cycles faster than sequential execution, while the other 5 invocations have short execution times, and TLS is 100 cycles slower. With the 6 invocations, TLS leads to an overall speedup of $2000 - 5 * 100 = 1500$ cycles. However, if deprived of cycle numbers, this loop could be considered a poor candidate because for 5 out of 6 invocations, TLS yields lower performance. This could lead to *premature serialization* of this profitable loop level. Identifying loops that could lead to overall speedup requires quantitative evaluation of the impact of speculative threads.

Furthermore, an effective mechanism should identify a set of loops that lead to maximum performance benefits. In many programs, it is common to have multiple nesting loop levels that all benefit from TLS. Since the optimal parallel loops are most likely neither the outermost nor innermost loop, finding the right set of loops is essential to maximize performance.

Finally, an effective mechanism should adapt to program phase changes. When a program enters a different phase, loop behaviors can change substantially. Loops that are previously serialized could potentially benefit from TLS in the new phase.

Therefore, it may be necessary to re-evaluate and re-select the best performing loops when phase changes. We describe our tuning mechanisms in four incremental policies, each extending the previous one with higher complexity.

4.2.2 Searching loop levels from inside out

The first policy follows an *inside-out* search order, i.e. from the innermost to the outermost, to evaluate the impact of TLS for each level in a loop nest. Each level runs in TLS mode for several invocations, and the number of times TLS performs worse than sequential execution is recorded in the saturation counter. Similar to branch prediction, once this counter exceeds a certain threshold, the loop is predicted as not suitable for TLS. This decision is stored in the decision table, so this loop will be serialized whenever it is encountered again, yielding resources to its outer loop levels. Otherwise, the current loop is continuously parallelized, and the search for the loop nest may stop at the current level. We name it **InsideOut** since it is the fundamental search order for other policies.

Although this policy could use the opposite *outside-in* search order instead, outer loops are much larger than inner loops and attempting outer loops would significantly prolong the time needed to reach the best level. In extreme cases, the outermost loop covers the entire program execution; by the time the outermost loop level is evaluated, the execution of the program is almost done.

4.2.3 Using quantitative evaluation

`InsideOut` is prone to serializing a loop prematurely, while the loop could lead to overall speedup. Loop invocations with different execution times should not be treated equally. We devise a new policy named **Quantitative** that quantitatively evaluates every loop invocation and uses the performance summary, i.e. cycles saved from sequential execution, as the weights of different invocations. Under this policy, a loop is serialized if both the saturation counter exceeds the threshold *and* the performance summary becomes

negative.

4.2.4 Incorporating static analysis through compiler annotations

Both `InsideOut` and `Quantitative` stop searching once a loop level that could benefit from TLS is reached. Since multiple loop levels can all benefit from TLS, previous searching mechanisms cannot guarantee reaching the level that has the highest performance benefit. Although an exhaustive search could find the best level, it takes too much time and is not cost-effective, as search time is part of run time. The new policy incorporates static analysis through compiler annotations to prioritize the search order. It starts from the loop level annotated by the compiler. If this level indeed improves performance, the search is over; otherwise, this loop level is serialized and the search begins from the innermost level. We name it **Quant+Static** as it adds static analysis on top of quantitative evaluation.

4.2.5 Overriding static analysis decisions

`Quant+Static` can potentially find the best level if static analysis is accurate. However, it can fall into the same sub-optimality if static analysis targets to the wrong loop level. Our final policy strives to protect against the situation where the compiler's choice is wrong. It explores both the compiler-annotated loop level and the neighboring levels and compares them quantitatively to increase the chance of reaching the best loop level. It is named **Quant+StaticHint** since we treat static analysis only as hints and could override them.

The full adaptation to phase change is beyond the scope of this dissertation. In our implementation, a simple mechanism is used for all the policies: the decision tables are reset periodically, so that the impact of speculative threads can be re-evaluated. Table 4.1 summarizes the features of the four tuning policies.

Table 4.1: Summary of the Performance Tuning Policies

Policy	Search Order	Performance Evaluation	Static Analysis
InsideOut	Inside out	Based on invocations	Not used
Quantitative	Inside out	Quantitative	Not used
Quant+Static	Inside out	Quantitative	Use as decision
Quant+StaticHint	Inside out	Quantitative	Use as hint

4.3 Performance Evaluation

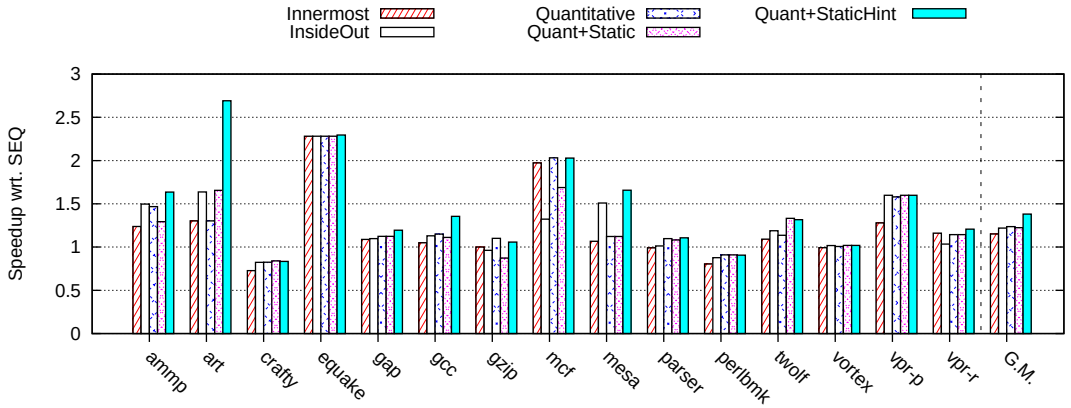
In this section, we will first compare the performance among the four tuning policies (Chapter 4.2), then evaluate how the different performance prediction schemes (Chapter 3.2) impact the performance of the tuning system. Finally, we will contrast our dynamic tuning mechanism with the state-of-the-art static thread management.

4.3.1 Impact of Dynamic Tuning Policies

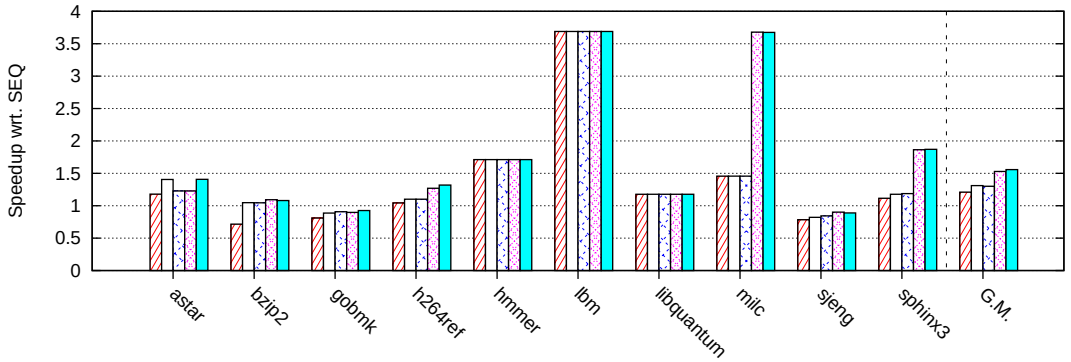
Figure 4.2 compares the speedup with respect to sequential execution among the four increasingly sophisticated tuning policies. In addition, the scheme that blindly parallelizes the innermost loop level (bar `Innermost`) is also included in the comparison. All speedup numbers are summarized by geometric mean (G.M.).

Innermost vs. InsideOut

Comparison between the first two bars indicates that `InsideOut` generally outperforms `Innermost`, however with the exception of GZIP, MCF, and VPR-R. A close examination of the execution traces reveals that `InsideOut` missed some profitable inner loop levels in these benchmarks due to premature serialization (Chapter 4.2.1), whereas `Innermost` persistently parallelizes the innermost loop levels regardless of their performance. However, for many other benchmarks, our first tuning policy greatly outperforms `Innermost` by a large margin, especially in AMMP, ART, MESA, VPR-P, ASTAR and BZIP2, showing its usefulness. `InsideOut` achieves an overall speedup of 1.255x, while `Innermost` achieves



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 4.2: Performance comparison of different dynamic tuning policies. Speedups are normalized to the original sequential execution (SEQ).

1.175x.

Simple vs. Quantitative

`Quantitative` weights different loop invocations by the cycles saved from sequential execution, and is more accurate than `InsideOut` in identifying profitable loop levels. It resolves the problem of premature serialization, as the performance is improved in GZIP, MCF, and VPR-R.

Yet ART, MESA and ASTAR experienced significant performance downgrade in this

new policy. This is due to the existence of multiple overlapping loop levels that are all profitable. `Quantitative` stops searching once a profitable level is found, without checking whether an even profitable one lies outside. We refer this problem as *local optimality*. `InsideOut` and `Quantitative` both suffer from it in a number of benchmarks. For the cases in ART, MESA and ASTAR, however, `InsideOut` prematurely serializes the inner loop level by mistake and reaches a more profitable outer level, making it outperform `Quantitative`. `Quantitative` has an overall speedup of 1.262x, slightly better than `InsideOut`.

Quantitative vs. Quant+Static

The `Quant+Static` policy incorporates compiler annotations to prioritize the search in the loop nest. We found that it selects better loop levels for benchmarks ART, TWOLF, H264REF, MILC and SPHINX3, but it greatly degrades AMMP, GCC, GZIP, and MCF. And the performance downgrade in MESA and ASTAR remains. The problem is that compiler annotation often fails to point to the most profitable loop levels. `Quant+Static` respects the compiler decision and does not attempt to look at other levels as long as the annotated loop level does not worsen performance. In another word, the problem of local optimality is still at large. Overall, this policy shows 1.338x speedup on average.

Quant+Static vs. Quant+StaticHint

The `Quant+StaticHint` policy treats compiler annotation only as hints and evaluates both the annotated loop level and its neighboring levels. The compiler-annotated loop will be compared with its inner loops. If an annotated loop level is the innermost, it will be compared with the immediate outer loop level. In both cases, this policy can select a loop level that outperforms the compiler's decision and try to avoid local optimality.

With the `Quant+StaticHint` policy, almost all benchmarks benefit. Inaccurate static loop selections are overridden in GZIP, MCF and ASTAR, so the performance becomes similar to `Quantitative`. At the same time, TWOLF, H264REF, MILC and SPHINX3

enjoys the benefit of accurate compiler annotations. More importantly, for AMMP, ART, GAP, GCC, MESA, and VPR-R, loop levels with higher performance are discovered and parallelized, which leads to better results than both `Quantitative` and `Quant+Static`. In Chapter 4.4, we will look into a few benchmarks as case studies to illustrate how this tuning policy searches the loop nest and to offer insights into why it outperforms static analysis.

Across all the benchmarks, `Quant+StaticHint` generally yields the best performance among all tuning policies, while negligible overhead is observed in some benchmarks due to tentatively trying out non-optimal loop levels for comparison. The average speedup to sequential execution is 1.449x.

Summary

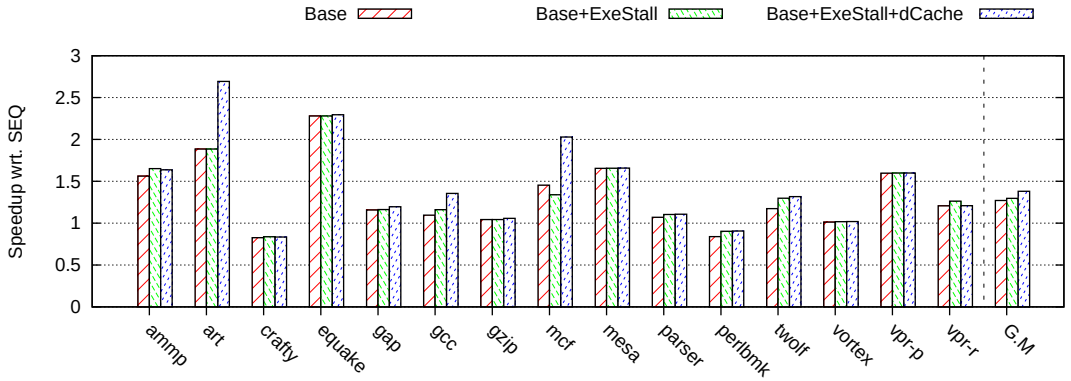
Performance gradually improves as we build up the desired tuning policy, rising from the most basic that achieves 1.255x to the most sophisticated that achieves 1.449x.

4.3.2 Impact of Performance Prediction Schemes

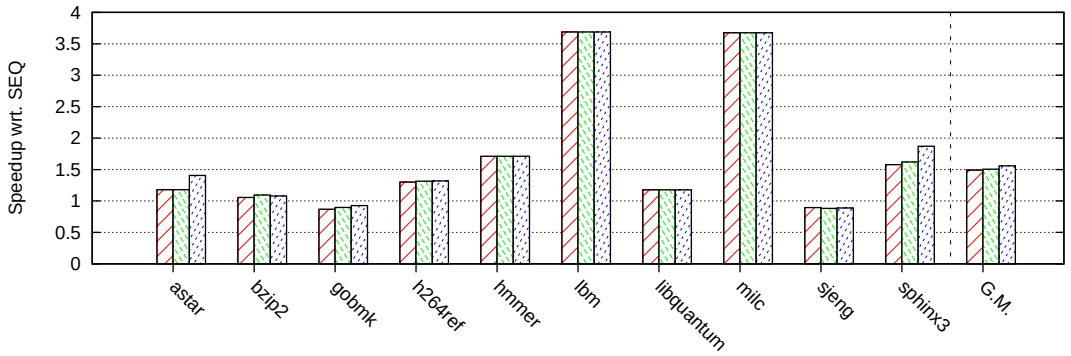
Chapter 3.2 has proposed three increasingly complex schemes to predict sequential execution and evaluated their accuracy. In this section, we will evaluate their performance impact for the tuning system. For the purpose of this comparison, we will use the most successful tuning policy `Quant+StaticHint`.

Figure 4.3 contrasts the performance of these prediction schemes. `Base` corresponds to the baseline prediction scheme described in Chapter 3.2; `Base+ExeStall` incorporates the execution stall scaling described in Chapter 3.2.1; and `Base+ExeStall+dCache` further incorporates the data cache behavior classification described in Chapter 3.2.2.

A general trend of increasing performance is observed as the prediction scheme is amended to model TLS execution in more details. `Base` yields the lowest performance because the inaccuracy in prediction causes the runtime system to often target on wrong loops. Execution stall scaling and data cache behavior classification prove to be useful in



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 4.3: Performance comparison of different prediction schemes. Speedups are normalized to the original sequential execution (SEQ). Note that bar `Base+ExeStall+dCache` is also the last bar in Figure 4.2.

correcting this inaccuracy. For example, in TWOLF incorporating execution stall scaling recovers most of the performance loss; while in ART, data cache behavior classification is key to prediction accuracy; nevertheless, GCC benefits from both amendments. MCF is a memory-bound benchmark. Only applying execution stall scaling happens to expose more inaccuracy; however, further incorporating data cache behavior classification ultimately corrects this error.

Benchmark VPR-R exhibits some abnormality as incorporating data cache behavior

classification results in lower performance. This is due to a simplification when predicting conflict misses in sequential execution. We will discuss this problem and its possible remedy in further detail in Chapter 4.3.3.

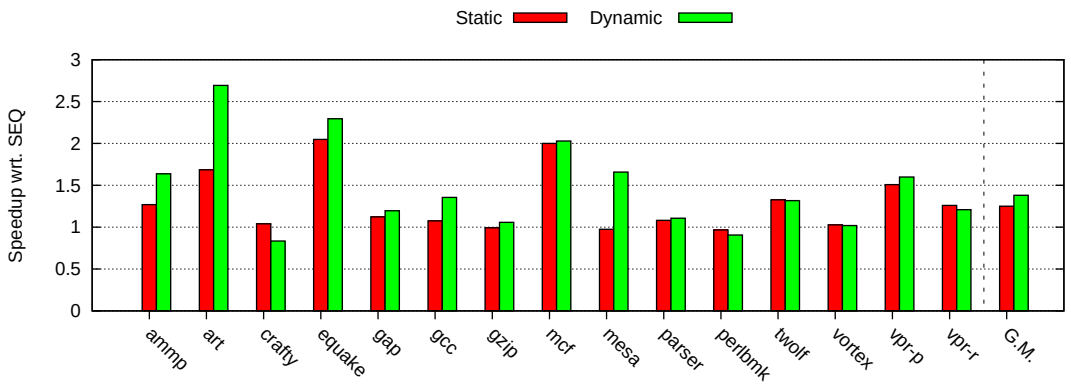
Overall, `Base` and `Base+ExeStall` show the speedup of 1.355x and 1.376x, a 6.5% and 5.0% loss compared to `Base+ExeStall+dCache` (1.449x), respectively.

4.3.3 Performance Comparison with the Static Approach

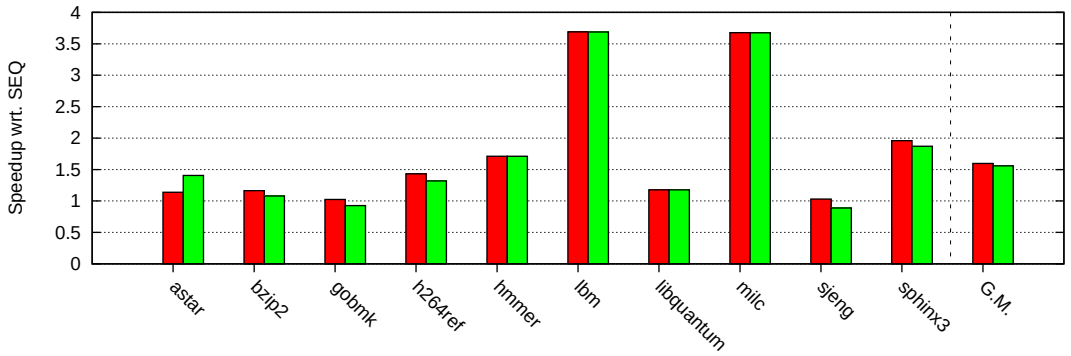
This section contrasts our most successful tuning policy (`Quant+StaticHint`) with the state-of-the-art static analysis. Before presenting a detailed comparison, we first explain the overhead introduced to TLS. To parallelize a loop, necessary special instructions (thread spawning, synchronization instructions such as signal and wait, and thread committing) are generated. These extra instructions are TLS overheads. For static approach, the compiler selects only a subset of loops to insert these instructions. But for dynamic approach, all the loops are instrumented so that any loop may be eligible for parallel execution. When the code is running in sequential mode, TLS-specific instructions are executed as NOPs (stands for no operation), but still incur extra performance penalty. If a program has a large number of small loops that are instrumented but eventually serialized, the slowdown due to this overhead can be significant. We referred to this as *parallel code overhead* for TLS.

One way to completely eliminate this overhead is to generate two versions of code for each loop, a parallel version and a sequential version, and switch between these versions as needed. However, we are unable to experiment with this scheme due to limitations in our infrastructure. To estimate the performance after eliminating this overhead, we attempt to normalize the TLS runs to the sequential execution of their respective parallelized code.

Parallel code overhead can also be mitigated in several ways. Using simple heuristics, the compiler or programmer can filter out some loops first. For example, loops with tight dependencies (such as pointer chasing or short reduction) are unlikely to benefit



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 4.4: Performance comparison against the state-of-the-art static approach. Speedups are normalized to the original sequential execution (SEQ). Note that bar `Dynamic` is also the last bar in Figure 4.2.

from TLS. A runtime re-optimization system can also eliminate such overhead for all the loops not selected for TLS execution. We do not explore these optimizations and leave their integration as future work.

In Figure 4.4, `Static` and `Dynamic` are both normalized to the execution time of the original sequential executable where no instrumentation is made. Over all the benchmarks, dynamic tuning (1.449x) outperforms static analysis (1.379x) by 5.1%.

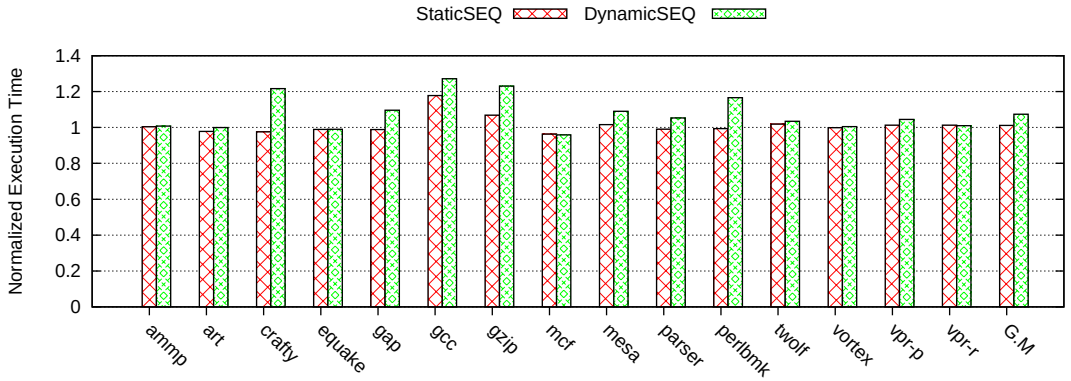
We have observed that SPEC CPU2000 and CPU2006 benchmark suites have different responses to dynamic tuning. Many SPEC CPU2000 programs are inherently

sequential and hence are difficult for the static analysis to identify the optimal loop levels. Dynamic tuning is able to correct most of the non-optimal decisions made by the compiler and improve performance by a significant margin, especially in AMMP, ART, EQUAKE, GCC, and MESA. Overall, dynamic tuning (1.381x) outperforms static analysis (1.251x) by 10.4%.

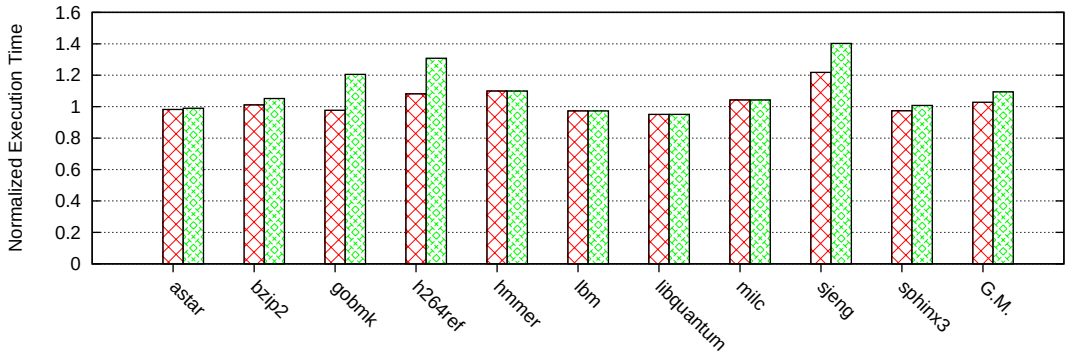
SPEC CPU2006 benchmarks generally exhibit more parallelism and in most cases the compiler is able to identify the optimal loop levels, and thus leaves little or no room for dynamic tuning mechanism to further improve the performance. There is one case, however, that in ASTAR static analysis selects a wrong loop level and it is corrected by dynamic tuning. Due to the high parallel code overhead, dynamic tuning (1.558x) is slight worse than static analysis (1.596x) by 2.4%.

There are a number of cases where dynamic tuning is worse compared to static analysis, such as in CRAFTY, PERLBMK, TWOLF, VORTEX, VPR-R, BZIP2, GOBMK, H264REF, SJENG and SPHINX3. We find that it is the parallel code overhead that causes most of the performance downgrade. We run the static and dynamic executables sequentially, and their execution time is shown in Figure 4.5 as bars `StaticSEQ` and `DynamicSEQ`, respectively. The higher the bar is, the greater the overhead is. In Figure 4.6, `Static/StaticSEQ` and `Dynamic/DynamicSEQ` are normalized to their corresponding sequential baselines. In this rescaled comparison, most previously under-performing benchmarks has shown comparable or even better performance under dynamic tuning. Overall, dynamic tuning (1.568x) could potentially outperform static analysis (1.404x) by 11.7%.

After the rescaling, VPR-R is the only benchmark in which dynamic tuning performs noticeably worse than static analysis. The reason is due to inaccurate performance prediction discussed in Chapter 3.3.2. For a profitable loop in VPR-R, our model does not predict the full extent of conflict misses that would have occurred in sequential execution. Therefore, sequential execution cycle is predicted less than what it should be, and also less than the TLS execution cycle, leading to an incorrect conclusion that this loop does not benefit from TLS and consequently, this profitable loop is serialized.



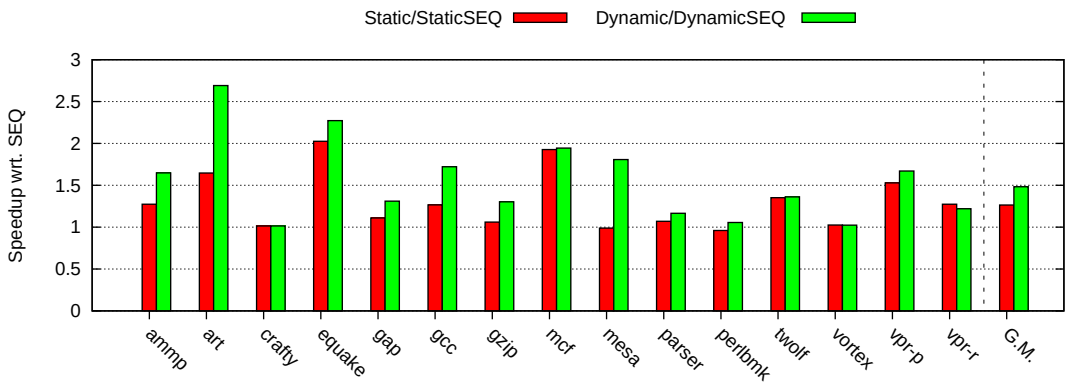
(a) SPEC CPU2000 Benchmarks



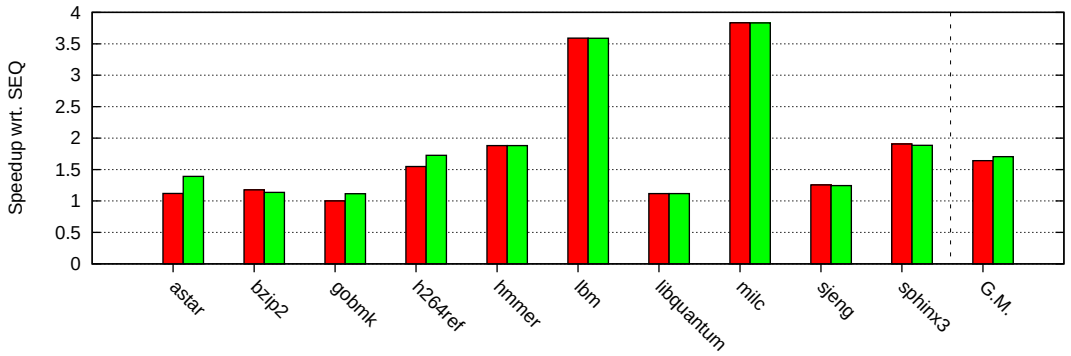
(b) SPEC CPU2006 Benchmarks

Figure 4.5: Execution time of the parallelized binary running sequentially. All instrumented loops will incur overhead. The higher the bar, the worse the overhead.

Static approach benefits from parallelizing this loop. We have observed significant count of conflict cache misses when this loop is serialized and run sequentially. This observation could lead to a potential remedy to this problem: if a serialized loop incurs high rate of conflict misses, we should give it another chance to be parallelized, taking into account the benefit of parallel execution to reduce conflict misses from sequential execution.



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

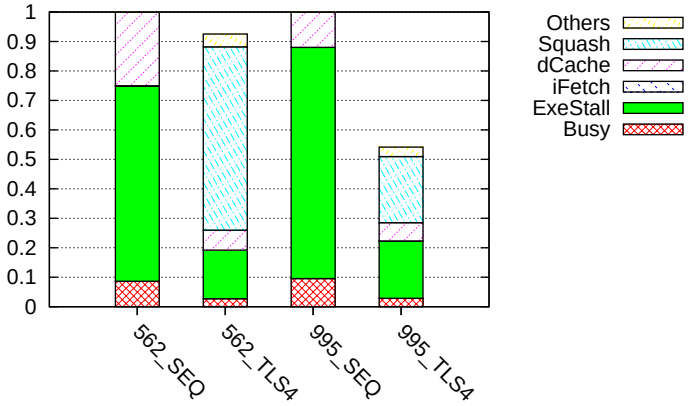
Figure 4.6: Performance comparison between static analysis and dynamic tuning when normalized to their respective sequential versions.

4.4 Case Studies

In this section, we examine into details on benchmarks where dynamic tuning performs much better than static analysis and explain why this can only be achieved through runtime decision-making. Appendix B will list out the different loops selected by static analysis and dynamic tuning for each benchmark. Many cases have similar behaviors, so we highlight in this section the cases of AMMP, ART and MESA.

```
562: for( inode = 0; inode < nx*ny*nz; inode ++)
     {/* loop over all mm nodes */
        ...
995:    for( i=0; i< imax; i++)
        {
           a2 = (*atomall)[i];
998:       for( j=0; j< a1->dontuse; j++)
           { if( a2 == a1->excluded[j]) goto SKIPNEW; }
                 ...
     SKIPNEW: j = j;
        }
     }
```

(a) AMMP code snippet in rectmm.c



(b) Performance characteristics of loops inside the snippet

Figure 4.7: Case study on AMMP

4.4.1 Benchmark AMMP

The performance improvement of AMMP from `Static` to `Dynamic` in Figure 4.4 comes from the different loop levels selected by them. One instance of such difference is located in source file rectmm.c. The outer loop starts at line 562 and the inner loop starts at line 995. We named loops by their starting line number. Their code snippets and execution time breakdowns are shown in Figure 4.7(a) and Figure 4.7(b). The bars are labeled with loop name and its execution mode. For example, 995_SEQ and 995_TLS4 correspond to loop 995 running sequentially and in parallel, respectively. Bars are normalized to sequential executions with respect to the same loop. The static analysis

believes the outer loop 562 has a greater performance benefit than the inner loop 995. However, loop 562 incurs frequent speculation failures (i.e., is squashed), which cannot be predicted by the compiler. Although the inner loop 995 would also incur speculation failure if selected for TLS, it achieves a better speedup than parallelizing loop 562.

Our dynamic dispatching policy `Quant+StaticHint` uses cycle-saving as the measurement. In this case, it works as follows: since the outer loop 562 is selected by the compiler, `Quant+StaticHint` policy first parallelizes inner loop levels within loop 562 at its first invocation, and innermost level loop 998 is parallelized for comparison. Since loop 998 degrades performance, it is quickly serialized. In the next step, loop 995 is parallelized and the cycle-saving is recorded. At the second invocation of loop 562, this compiler-selected loop is parallelized while all of its inner levels are tentatively serialized and the cycle-saving is also recorded. From the third invocation on, loop 562 and loop 995 are compared and the one with greater cycle-saving is selected for TLS execution. In this case, loop 995 wins the competition.

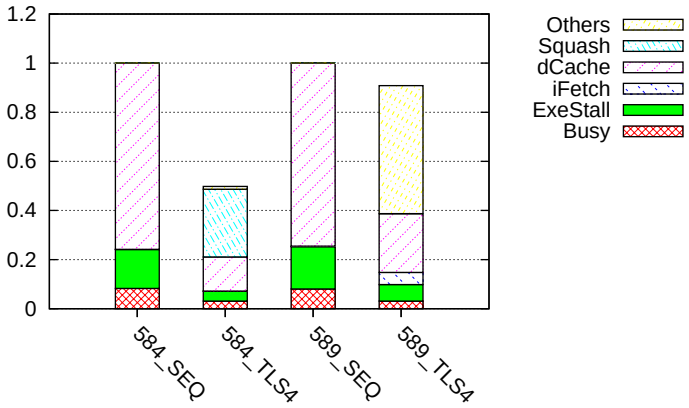
4.4.2 Benchmark ART

Differences in the performance of ART, as shown in Figure 4.4, partly come from source code in scanner.c: static analysis chooses the inner loop starting at line 589, whereas dynamic mechanism favors the outer loop starting at line 584. Figure 4.8(a) and Figure 4.8(b) show the code snippets of these two loops and contrast their performance. Bars are labeled as previously.

Judged by the execution time breakdown of 584_TLS4 alone, this loop should not be parallelized since the cost of speculation failure (the `Squash` segment) is high. However, when compared against the sequential execution 584_SEQ, the failed speculative threads help to fetch useful data into the L2 data cache and reduce the data cache stalling (`dCache`) to a large extent. While this cache effect (discussed in Chapter 3.2.2) benefits parallel execution, its impact is hard to accurately estimate at compile time. Our compiler uses dependence profiles to estimate speculation failures, so loop 584 is

```
584: for (ti=0;ti<numf1s;ti++) {
        tsum = 0;
        ttemp = f1_layer[ti].P;
589:    for (tj=0;tj<numf2s;tj++) {
           if ((tj == winner)&&(Y[tj].y > 0))
              tsum += tds[ti][tj] * d;
        }
        f1_layer[ti].P = f1_layer[ti].U + tsum;
        tnorm += f1_layer[ti].P * f1_layer[ti].P;
        if (ttemp != f1_layer[ti].P)
           tresult=0;
     }
```

(a) ART code snippet in scanner.c



(b) Performance characteristics of loops inside the snippet

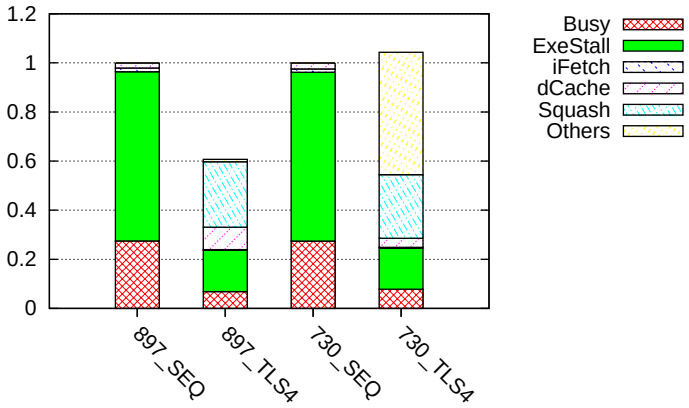
Figure 4.8: Case study on ART

determined not ideal for TLS due to possible speculation failures from runtime aliasing. This is why the static analysis chooses the inner loop 589 for TLS. Unfortunately, the performance of inner loop 589 is not up to the expectation due to the smaller coverage and *insufficient thread count* (part of `Others` in the breakdown). Insufficient thread count means the total number of thread is less than the number of available processing cores (four in our experiment), so some cores are left idling when this loop is being executed. This contributes most of the `Others` segment in the breakdown. Dynamic dispatching policy is able to compare the compiler-selected inner loop 589 with the outer loop 584 and ends up selecting loop 584 for better performance gain.

```
897: for (i=2;i<VB->Count;i++) { /* vrender.c */
       START_PROFILE
       (*ctx->Driver.TriangleFunc)( ctx, i-2, i-1, i, i );
       END_PROFILE( ctx->PolygonTime, ctx->PolygonCount, 1 )
     }

471: static void general_textured_triangle /* triangle.c */
     {
       ...
730: while (lines>0) { /* tritemp.h */
         GLfixed ffz = fz;
         ...
       }
       ...
     }
```

(a) MESA code snippet in vrender.c, triangle.c and tritemp.h



(b) Performance characteristics of loops inside the snippet

Figure 4.9: Case study on MESA

4.4.3 Benchmark MESA

The performance difference between static and dynamic approach mainly comes from the code snippet shown in Figure 4.4.3. In vrender.c, an outer loop at line 897 calls a function that contains an inner loop at tritemp.h:730. The execution breakdowns of these two loops are shown in Figure 4.9(b).

Static analysis chooses the inner loop 730, which turned out to have a low thread count per loop invocation. Thus, TLS execution is unable to utilize all the cores efficiently. It is shown in the breakdown as the `Others` segment is significant. This also

correlates with the loop coverage in Figure 3.5 that most loops selected by the static approach perform no better than 5% compared to their sequential execution. Dynamic dispatching approach attempted to parallelize both loops, and found it more beneficial to parallelize the outer loop. We noticed that `dCache` segment increases from sequential to parallel execution for loop 897. This is because parallel execution needs to load the same data item to multiple L1D caches while sequential execution for the same code segment may need to load only once. Nevertheless, this minor overhead does not offset the benefit of speculative parallelization in this case.

4.5 Related Work

Most prior works on TLS rely on compiler to extract speculative threads. Among these approaches, the POSH [21] TLS compiler partitions the program into tasks based on code structures like loops and subroutines. It uses a simple profiling pass for weeding out the ineffective tasks. The Mitosis compiler [19] inserts spawning pairs to transform a sequential program into a number of speculative threads. The selection of spawning pairs is also dependent on static analysis and profiling information. Wang *et al.* [20] and Du *et al.* [17] use extensive profiling information to statically estimate loop performance and select a set of loops for parallelization to maximize overall program performance. Johnson *et al.* [18] propose the balanced min-cut approach to decompose the program at compile time. They use a graph theory based framework to partition the program taking into account overheads of data dependence, load imbalance, and thread prediction. Vijaykumar and Sohi [15] select the proper tasks for speculation in Multiscalar [6] using compiler heuristics based on characteristics such as task-size, inter-task control flow, and data dependences. These compiler-based approaches, by virtue of being static, cannot predict program behavior accurately. Program behavior depends on numerous factors such as micro-architectural features, inputs applied, as well as memory access patterns, none of which can be accurately projected beforehand. Our approach analyzes

the program execution at runtime and can naturally adapt to these variations.

There is a large body of works on dynamic program optimization [25, 26, 27, 28, 29, 30]. For example, DynamoRIO [26, 28] uses a combination of a native Just-In-Time compiler and partial evaluation techniques. Lu *et al.* [30, 29] generate helper thread prefetches using information obtained from the hardware monitors on the Sun UltraSPARC®. The optimization framework proposed in ADORE [30, 29] is similar to the speculative thread optimization framework in this chapter, with the following differences: (i) our work uses hardware-based performance counters that generate cycle breakdowns [59, 60], while ADORE uses event-based hardware performance counters; (ii) in ADORE, a dynamic compiler is responsible for generating and patching re-optimized code at runtime, while our scheme does not rely on dynamic code generation; and (iii) we carefully evaluate the performance impact of speculative threads before optimization, while ADORE does not evaluate the effectiveness of the prefetching threads.

Optimizing for OpenMP [31] workloads dynamically has been explored by many researchers [32, 33, 34, 35]. stOMP [33] selects among multiple specialized versions of parallel regions based on parameters; Zhang *et al.* [35] experiment with different OpenMP scheduler configurations at different parallel regions. Lee *et al.* [34] peel parallel loops and collect performance profiles using the first few iterations of the loop to re-optimize the program dynamically. However, performance optimization for OpenMP is very different from that for speculative threads. First, the tuning parameters are different. For OpenMP, the tuning parameters are the number of threads [32], shared variables [33], and etc., whereas our system optimizes TLS performance by re-deciding where to speculate. Second, the performance models for OpenMP and TLS differ significantly: in OpenMP, all threads perform useful work, while in TLS, work done by speculative threads can be wasted when speculation fails. The knowledge learned from OpenMP performance optimization cannot be directly applied to TLS. However, we believe that our performance analysis technique (cache behavior classification, execution

stall scaling, and etc) and dynamic tuning policy can be applied to OpenMP threads to understand their performance.

Kim *et al.* [61] proposed a software runtime system that enables speculative parallelization techniques on non-shared memory clusters. The large speedup reported (49x on average) is primarily due to the large number of available cores (128 in total) and the use of speculation techniques that efficiently address inter-node communication costs. In contrast, our research is built on a TLS-enabled 4-core CMP system with cache-coherent shared memory.

4.6 Summary

In this chapter, we have presented an execution framework that facilitates dynamically monitoring, evaluating, and adjusting the behavior of speculative threads. Hardware-based performance counters are programmed to collect necessary runtime information needed to evaluate the efficiency of speculative threads. We have proposed performance tuning policies that aim to search for the best spawning points for speculative threads based on their efficiency.

Combining the most sophisticated tuning policy and the most accurate prediction model, our dynamic tuning system is able to outperform the state-of-the-art static analysis by 5.1% on average over a large set of programs from the SPEC CPU2000 and CPU2006 benchmark suites. Compared to the original sequential execution, our dynamic tuning system has an average speedup of 1.449x.

In summary, we believe that dynamic performance tuning proves to be effective in extracting speculative thread-level parallelism from sequential programs. We have also learned that quantitative evaluation of speculative thread efficiency and incorporation of static analysis are the two key ingredients to a successful tuning policy.

Chapter 5

Heterogeneous Multicore Design Space Exploration

Thread-Level Speculation (TLS), by optimistically ignoring infrequent memory aliasing, could improve performance and save power if the success rate is high. On the other hand, failed speculation could lead to prolonged execution time and increased power consumption, potentially degrading energy efficiency. In this chapter, we will look into the possibility of incorporating on-chip heterogeneous components to improve energy efficiency, measured in energy-delay-squared product (ED^2P).

We will first discuss some important design issues, and then build an idealistic execution environment to explore the design space. Based on the exploration, we will present a feasible design solution and estimate its upper-bound in energy efficiency improvement. A case study will show some interesting code behaviors and their performance and energy characteristics.

5.1 Design Issues

Matching the performance characteristics with the appropriate hardware configuration is key to the success of the heterogeneous proposal. Below we discuss the forms of

on-chip heterogeneity we intend to explore and some general guidelines in choosing an appropriate configuration.

5.1.1 Forms of heterogeneity

Integration of heterogeneous cores onto a single die has been proposed previously for executing sequential [36] and parallel workloads [37]. These techniques, however, cannot be directly translated to the speculative-thread environment. We indent to consider a set of same-ISA cores with different computing power (i.e. issue width) and with or without hardware support for Simultaneous Multi-Threading (SMT).

Cache configurations have significant impact on energy efficiency. The cache components can configured in terms of total size, number of sets and associativity. Reconfiguring cache for speculative threads exhibits complexities that do not exist in sequential programs. In this dissertation, we only focus on the impact of first-level cache(s), and hold second-level cache size (which is also the last-level cache on-chip) constant. This is because all cores share the same second-level cache in our system; and the aggregated memory accesses generated by all speculative threads are similar to that generated by the corresponding sequential program. Thus, altering the second-level cache size will have similar impact on performance whether or not the program is speculatively parallelized.

TLS can be supported on processors with different forms of multi-threading: a single core with SMT support, as well as multiple independent cores (CMP) each executing a single thread. Previous work has shown that SMT and CMP could result in different levels of energy efficiency [62]. Thus, we explore the possibility for both types of multi-threading support.

5.1.2 Resource allocation guidelines

Speculative threads exhibit a number of unique sharing and contention patterns, which pose as both challenges and optimization opportunities for our design. The resource

allocation guidelines for each case are summarized in Table 5.1.

Trade-off between instruction-level and thread-level parallelism

When a program executes without being speculatively parallelized, instruction-level parallelism (ILP) can be extracted from the original sequential execution trace. When parallelized with speculative threads, each thread carries a much shorter instruction trace, which in turns leaves less instruction-level parallelism for the processor to exploit. In this case, smaller superscalar cores or a single SMT core are preferred.

Data spread across multiple L1 caches

Speculative threads extracted from sequential programs often share data at a fine granularity. When parallelized across multiple superscalar cores, each with its own private L1 cache, the average L1 cache miss rate can increase due to the fact that data must be loaded into multiple L1 caches. These additional cache misses do not occur when parallel threads execute on a single core that supports SMT. Therefore, SMT is preferred when data is shared at a fine granularity.

Prefetching effect between speculative threads

Data brought into the shared cache by speculative threads can also be used by other speculative or non-speculative threads, thus eliminating cache misses that would otherwise occur. One difference between speculative threads and fully parallel threads is that speculative threads have an implied order while parallel threads may be executed in any order. Due to the ordering constraint, speculative threads have been particularly effective in prefetching data into the shared cache for the subsequent threads. Even a failed speculative thread may still improve performance without degrading energy efficiency.

Table 5.1: Guidelines for resource allocation

Guideline I	Prefer smaller superscalar cores or an SMT core when ILP is low.
Guideline II	Prefer SMT when data is shared at a fine granularity.
Guideline III	Speculative failure can be benign if it prefetches data.
Guideline IV	Prefer CMP if processor resources are highly contended.
Guideline V	Carefully manage resources wrt. the reconfiguration overhead.

Resource contention among speculative threads

Speculative threads share resources when executed in a SMT processor. If all the threads are computationally intensive, the resource contention can stifle performance that can be otherwise achieved in a CMP. This situation can be exacerbated when speculative threads compete for resources but eventually fail and commit no useful work. Thus, we should favor CMP if resources are highly contended among speculative threads.

Frequent code segment interleaving with varying performance characteristics

TLS, taking advantage of fast on-chip communication latency, can potentially improve performance even when parallel threads are very small. This phenomenon leads to the frequent interleaving of parallel and sequential segments; as well as parallel segments with different performance characteristics. Consecutive code segments may require completely different architecture configuration to reach their best energy efficiency; however, the reconfiguration overhead can completely eliminate the benefit. Thus, we must carefully manage thread allocation and thread migration.

5.2 Design Space Exploration

Heterogeneous systems typically have a large design space. In this section, we first present an idealistic execution environment that allows us to efficiently experiment with a large number of hardware configurations. We will measure the usage of each component and estimate the upper-bound in energy efficiency.

Table 5.2: Architecture parameters scaled by the issue width

Core Type	1-Issue	2-Issue	4-Issue	6-Issue	8-Issue
Issue Width	1	2	4	6	8
Fetch Width	3	4	6	9	12
Commit Width	1	2	4	6	8
ROB Size	32	64	128	192	256
LSQ Size	24	32	64	96	128
Instr. Fetch Queue Size	8	16	32	64	64
Integer Units	1	2	4	6	8
Floating Point Units	1	3	4	5	6
Branch Predictor L2 Size	512	1024	1024	2048	2048

5.2.1 Experiment set-up

To explore the three forms of heterogeneity, we construct an execution environment with the following computing and caching components: (i) 1-, 2-, 4-, 6- and 8-issue cores with and without SMT support; (ii) 1-, 2-, 4- and 8-way associative first-level caches of size 16KB, 32KB, 64KB, 128KB, and 256KB. Note that the second level cache size is not a variable in our study. Architectural parameters of these cores are summarized in Table 5.2. Our methodology in determining these numbers is to start with a reasonable configuration of the four issue core (SimpleScalar default) and then scale them to other cores according to their respective issue widths. The L1 cache size varies by both the number of sets and the associativity, i.e. the same cache size would vary from 1-way to 8-way set-associative with a fixed 32-byte block size. Such variations apply to both data and instruction caches. The rest of the architectural parameters can be found in Table 2.1.

During program execution, we refer to a sequence of dynamic instructions that is speculatively parallelized as a *parallel segment*; and instructions sequentially executed between two parallel segments as a *sequential segment*. We limit the number of speculative threads in parallel segments to *four*, since previous work [11] has shown limited scalability beyond four speculative threads for applications in SPEC CPU benchmark

suite. In a parallel segment, speculative threads can either be allocated to one SMT core (SMT mode), or to multiple non-SMT cores each executing one thread of execution (CMP mode); however, they cannot be spread across multiple cores each running multiple SMT threads (mixed mode). We omit the mixed-mode execution from our experiment to avoid the complexity involved in maintaining speculation states across two types of multi-threading supports.

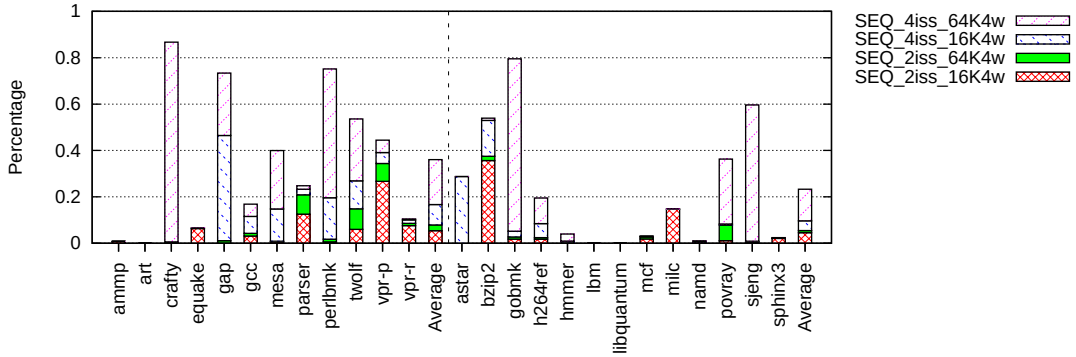
This execution environment is *idealistic* and we have made the following set of assumptions: (i) there can be as many cores and caches for each type as requested and all are connected through a bus; (ii) powering on and off cores and caches incur neither performance nor energy overhead; (iii) cores and caches are immediately powered-off after usage; (iv) cache contents become available immediately after the cache is powered on (no warm-up cost); and (v) for each segment of execution, an *oracle* is able to correctly predict and then use the hardware configuration that is most energy-efficient. The following predictions are made: core issue width, L1 cache configuration (associativity and size), and to use SMT or CMP for parallel segment. In Chapter 6, we will evaluate our proposal with these assumptions removed.

5.2.2 Heterogeneous component usage measurement

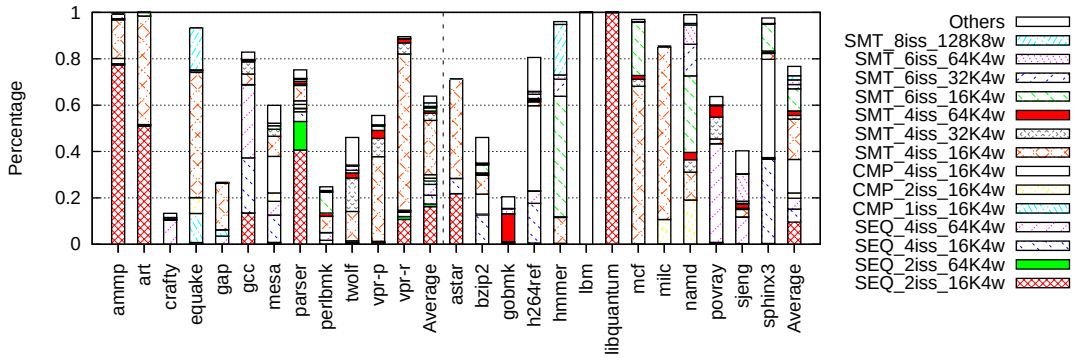
We first measure how often each configuration is used. Figure 5.1 shows, for each benchmark, the configuration usage breakdowns of both sequential and parallel segments. The stack segment corresponds to the percentage of execution time in which a particular configuration is activated by the oracle, i.e. achieving the highest energy efficiency. Configurations corresponding to less than 5% in all benchmarks are grouped in `Others`.

Diverse configuration preferences are observed not only among different benchmarks, but also within the same benchmark. We will highlight two cases to illustrate some of the key insights.

Benchmark LBM contains a high coverage loop (lbm.c:186) with high degree of both



(a) Sequential Segments



(b) Parallel Segments

Figure 5.1: Configuration usage breakdown of SPEC CPU2000 and CPU2006 benchmarks. Configurations prefixed with "SEQ" in parallel segments indicate that segment is most energy efficient if executed sequentially. "4w" means 4-way set-associative cache.

thread-level and instruction-level parallelism. If speculative threads from this loop were allocated to a single core in SMT mode, resource contention among threads would stifle the performance. This benchmark achieves the best ED^2P on four high-issue-width superscalar cores in CMP mode (Guideline IV).

In LIBQUANTUM, the performance bottleneck is high L2 cache misses. For a high coverage loop (gates.c:89), L2 miss rate is as high as 30%, and 85% of the processor cycles are stalled due to cache misses. CMP-based configurations with high-issue-width

superscalar cores are clearly inefficient. SMT-based configurations can potentially tolerate cache miss latencies by running multiple threads in parallel. In this case, however, with only four threads, SMT mode is unable to hide all L2 cache miss latencies. With respect to L1 cache size, the loop has very little reused cache blocks, and thus cannot benefit from a large cache. It turned out that the most energy-efficient configuration is a low-issue-width superscalar core with small L1 cache due to the low ILP (Guideline I).

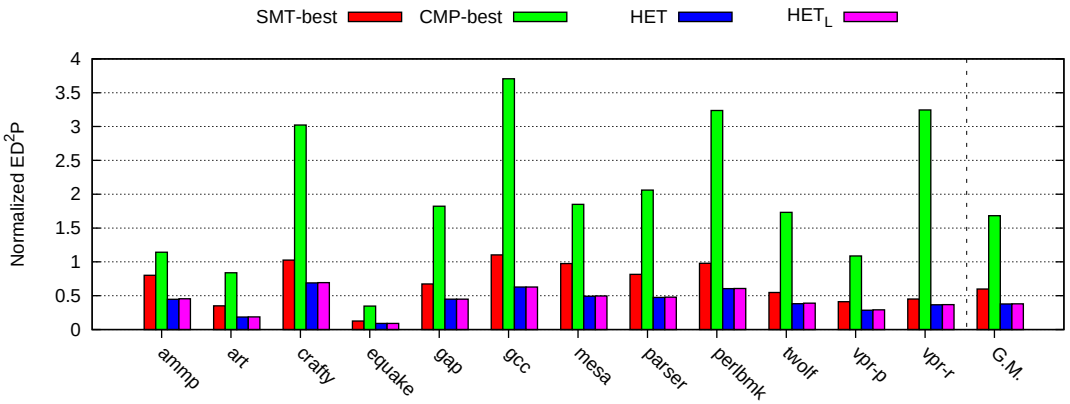
It is worth pointing out that, 4-way set associative cache is almost always the most energy-efficient choice for both sequential and parallel execution. Reducing associativity is particularly harmful for speculative parallel threads, because these threads utilize cache ways to buffer speculative writes [13]. Reducing cache associativity can increase the chances for speculative writes to overflow the cache, and in turn cause speculation to fail.

5.2.3 Energy efficiency upper-bound estimation

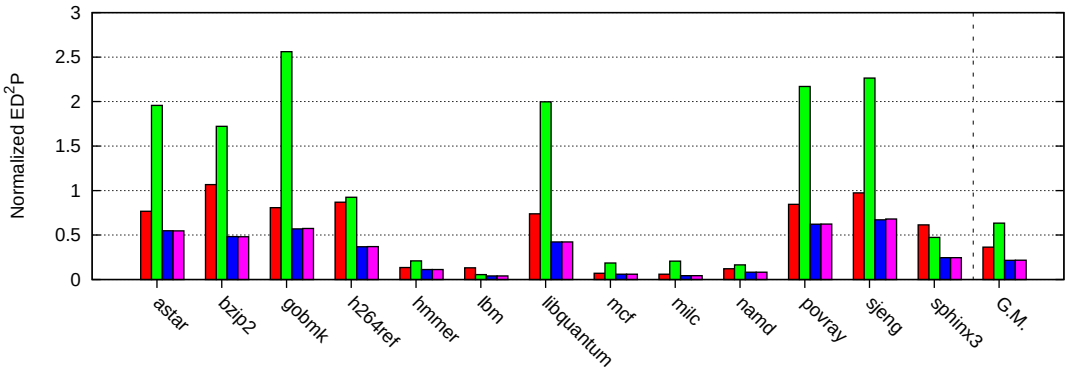
Using the idealistic execution environment, we can estimate the upper-bound of energy-efficiency improvement compared to *homogeneous* designs.

We measure energy efficiency on a variety of SMT-based and CMP-based homogeneous systems that also vary in the same core issue widths and L1 cache configurations. Figure 5.2 shows the best of the SMT-based ones (bar **SMT-best**, 4-issue core with 64K cache) and the CMP-based ones (bar **CMP-best**, four 4-issue cores each with 64K cache). Bar **HET** stands for the heterogeneous execution environment. The normalization base is unmodified code running on one 4-issue *SMT* core with 64K 4-way cache. Unless otherwise mentioned, this normalization base also is used throughout Chapter 5 and 6.

It is observed that even the best homogeneous systems sometimes degrade energy efficiency compared to the normalization base, e.g. BZIP2 for SMT and GCC for CMP. The heterogeneous environment is always the most energy-efficient one. Across all the



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 5.2: Upper-bound estimation wrt. the best homogeneous systems

benchmarks, we have observed the upper-bound of 38.8% and 72.1% in ED^{2}P reduction, compared to the best SMT-based and CMP-based homogeneous systems, respectively. The reduction compared to the normalization base is 71.7%.

Local vs. Global Decisions

Previous study [63] has shown that when performing dynamic optimizations for ED^{2}P, the best decision that minimizes ED^{2}P for a fraction of the total execution (local decision) may not lead to global reduction in ED^{2}P. Thus, dynamic optimizations that aim to reduce ED^{2}P face a fundamental difficulty: optimization decisions must be made

with knowledge or information of the future. Fortunately, this has little impact in our study. We have performed one extra experiment that makes locally optimal decisions (bar $\mathbf{HET_L}$). The previous experiment `HET` makes globally optimal decisions (assuming the future is similar to the past). We found negligible differences between these two experiments across all benchmarks (0.4%). Therefore, for the rest of Chapter 5 and 6, we allow the runtime system to make local decisions to optimize ED^2P.

5.3 A Feasible Design Solution

Obviously a feasible solution could not integrate all the used combinations from the design space. Thus it is crucial to identify a subset that (i) can capture most of the potential improvement and (ii) is feasible to integrate. Fortunately, the component usage measurement (Figure 5.1) can give us many insights.

Across all the benchmarks, CMP-based configurations are activated for 11.4% of the total execution time; and SMT-based configurations are activated for 36.3%. Thus, both execution modes should be supported. Second, configurations with 2-issue core(s) cover 22.0% of the total execution time; and configurations with 4-issue core(s) cover 65.8%. Although 6-issue core(s) has coverage of 9.1%, we found that the difference in energy efficiency is very small between 4- and 6-issue core(s). Thus, we should only integrate 2-issue and 4-issue cores. Third, configurations with 16K, 32K and 64K 4-way cache(s) represent 66.2%, 4.1% and 26.8% of total execution time, respectively. Other cache configurations correspond to only 2.9%. Thus, we should integrate 4-way associative caches, varying from 16K to 64K in size.

5.3.1 Proposed heterogeneous architecture

Figure 5.3 shows how these components are integrated. A *processing block* consists of one 4-issue SMT core, one 2-issue non-SMT core, and one 64K 4-way L1 cache. The two cores share the L1 cache through a switch [64]. The 64K L1 cache can be

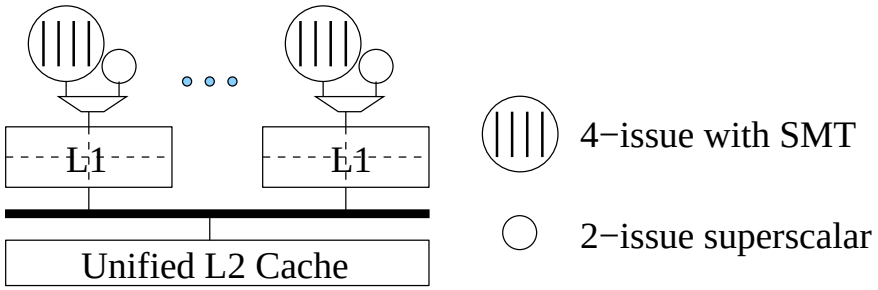


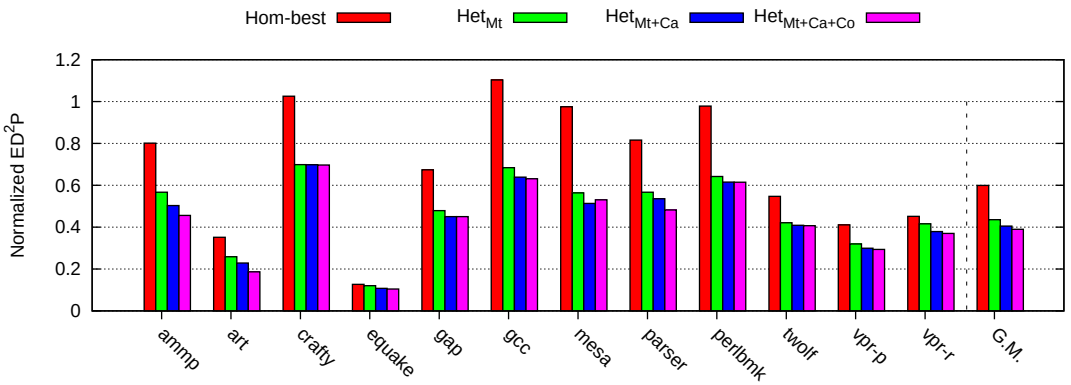
Figure 5.3: Proposed heterogeneous multi-core architecture for TLS. Each processing block consists one 4-issue SMT core, 2-issue non-SMT core, and re-sizable L1 cache.

dynamically reconfigured to 16K and 32K in size. Because the 4-way associativity needs to be maintained, we proposed to adjust the size by turning off part of the sets. A number of processing blocks are connected to the unified L2 cache. In this paper we limit the number of speculative threads to four, so each SMT core supports four threads of execution and four processing blocks are integrated. And we use bus as the interconnection media.

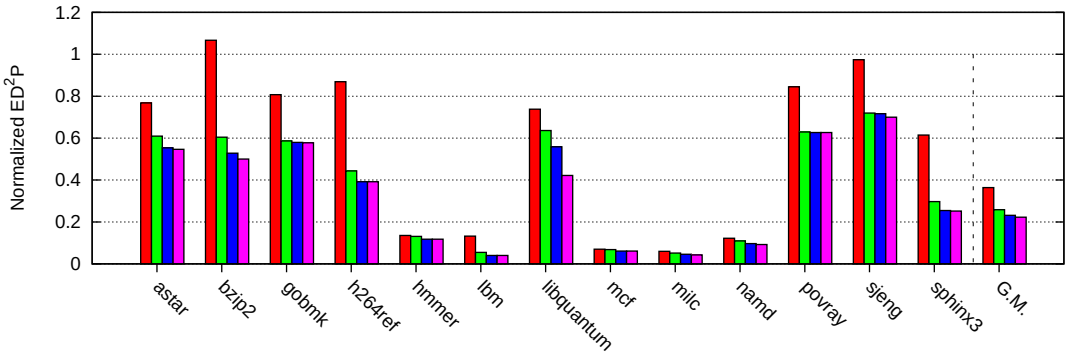
In this design, speculative threads can share one of the 4-issue cores (SMT mode); or spread across multiple 2-issue or 4-issue cores, each running one thread (CMP mode). Sequential segments can choose between one of the 2-issue and 4-issue cores. Note that the 4-issue SMT core will sometimes execute a single thread; this incurs some static power overhead that would otherwise not occur if the single thread executes on a non-SMT 4-issue core. However, we choose not to integrate the later so as to simplify the design and increase resource reusability so that more processing blocks may be integrated under a fixed transistor budget.

5.3.2 Understand where energy efficiency improvement comes from

We now attempt to estimate the energy efficiency for this feasible design and also understand why it improves energy efficiency compared to the homogeneous systems. We conduct our analysis by breaking the construction of our proposed architecture into



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 5.4: Energy efficiency improvement comes from the three forms of heterogeneity: multi-threading type, core issue width and cache size. Note that bar `Hom-best` is bar `SMT-best` from Figure 5.2.

incremental steps. In each step, we add one more form of heterogeneity to the best homogeneous system and measure the improvement to energy efficiency.

The best homogeneous system is a 4-issue SMT with 64K cache, shown as bar **Hom-best** in Figure 5.4. Here we assume that an SMT core consumes the same amount of energy as a non-SMT core when executing a single thread.

Multi-Threading Type: Bar **Het$_{\mathbf{Mt}}$** corresponds the system that replicates `Hom-best` to supports both SMT-mode and CMP-mode execution for speculative threads. Almost all the benchmarks are able to benefit from it significantly. ED^{2}P is reduced by 28.3%

compared to `Hom-best`.

L1 Cache Size: Bar $\mathbf{Het_{Mt+Ca}}$ builds on $\texttt{Het}_{\text{Mt}}$, and extends it with the ability to resize L1 cache between 64K and 16K. Plenty of benchmarks, such as AMMP, ART, GAP, ASTAR, BZIP2, and LIBQUANTUM, have shown noticeable improvement in energy efficiency. ED^2P is further reduced by 8.7%.

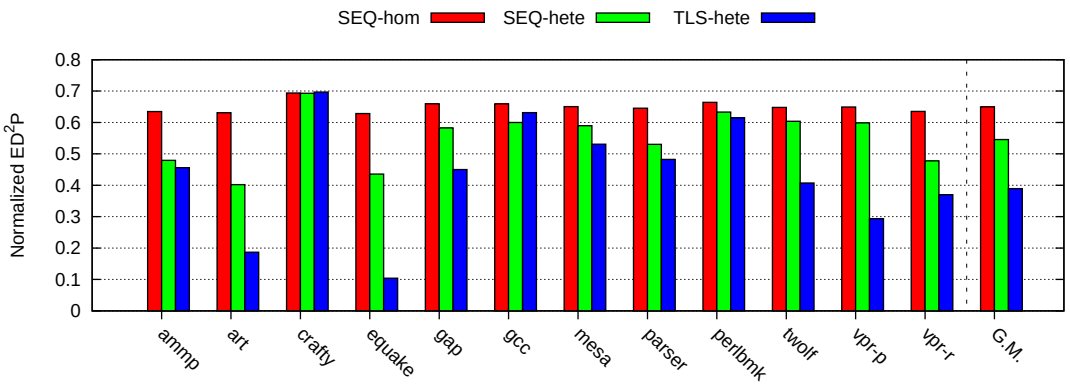
Core Issue Width: Bar $\mathbf{Het_{Mt+Ca+Co}}$ builds on $\texttt{Het}_{\text{Mt+Ca}}$, and augments each 4-issue core with a 2-issue core. Benchmarks such as AMMP, ART, PARSER, BZIP2, and LIBQUANTUM show noticeable additional improvement in energy efficiency. ED^2P is further reduced by 3.8%.

Overall, our proposed heterogeneous architecture (bar $\texttt{Het}_{\text{Mt+Ca+Co}}$) can potentially improve energy efficiency by 37.0%, a close match to the upper-bound of 38.8%, compared to the best homogeneous system. Thus the selection for the heterogeneous components is judicious.

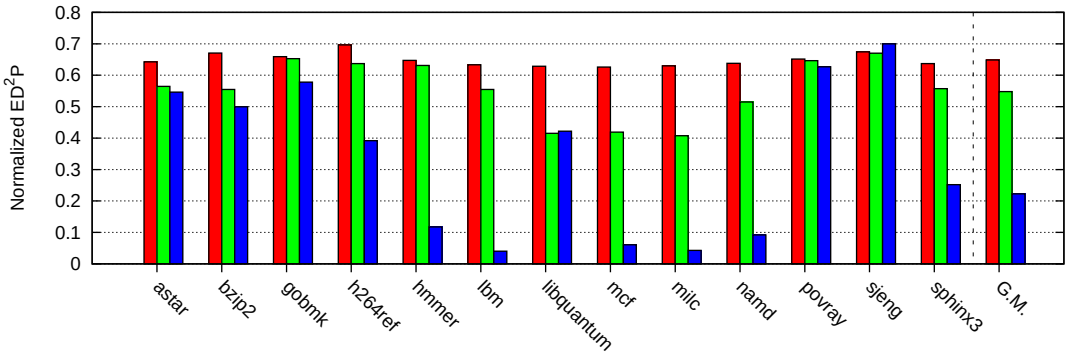
5.3.3 Comparison with a heterogeneous sequential processor

We are also interested to know how much the proposed TLS heterogeneous architecture could improve over a sequential processor that can adapt with the same choices in core issue width and L1 cache capacity. We have constructed another idealistic execution environment for executing the sequential version of the same benchmarks. The same set of assumptions is used as in Chapter 5.2.1. The configurations in this environment are the same as in the heterogeneous proposal. In other words, the core issue width can adapt between 2-issue and 4-issue and cache capacity between 64K-4way and 16K-4way.

Figure 5.5 shows the comparison. Bar **SEQ-hom** is the sequential homogeneous baseline with one 4-issue non-SMT core and 64K-4way L1 cache. It achieves the highest overall energy efficiency among other configurations that execute the sequential version of the benchmarks. (Note that the normalization base is sequential code running on a 4-issue SMT core, which incurs more static power consumption.) Bar **SEQ-hete**



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 5.5: Comparing with a heterogeneous sequential processor that can adapt with the same choices in core issue width and L1 cache capacity. Note that bar `TLS-hete` is also the last bar in Figure 5.4.

represents the potential energy efficiency of the heterogeneous sequential processor. Bar **TLS-hete** represents the potential energy efficiency of proposed heterogeneous TLS system.

Except for a few abnormalities where the thread-level parallelism is limited, `TLS-hete` performs significantly better than `SEQ-hete` in most benchmarks. Overall, the heterogeneous sequential processor improves energy efficiency over the sequential baseline by 15.7% (`SEQ-hete` vs. `SEQ-hom`), while our proposed heterogeneous system further improves over the former by 46.8% (`TLS-hete` vs. `SEQ-hete`).

```
12:  for (k=0; k<npairi; ++k) {
13:     const int j = pairlisti[k];
14:     register const CompAtom *p_j = p_1 + j;
     ...
69:     int jfep_type = p_j->partition;
70:     BigReal lambda_pair = lambda_table_i[2*jfep_type];
71:     BigReal d_lambda_pair = lambda_table_i[2*jfep_type+1];
     ...
97:     reduction[pairVDWForceIndex_X] -= 2.0 * vdw_dir * p_ij_x;
98:     reduction[pairVDWForceIndex_Y] -= 2.0 * vdw_dir * p_ij_y;
99:     reduction[pairVDWForceIndex_Z] -= 2.0 * vdw_dir * p_ij_z;
     ...
142:    Force & fullf_j = fullf_1[j];
143:    register BigReal tmp_x = fullforce_r * p_ij_x;
144:    fullElectVirial_xx += tmp_x * p_ij_x;
145:    fullElectVirial_xy += tmp_x * p_ij_y;
146:    fullElectVirial_xz += tmp_x * p_ij_z;
147:    fullf_i.x += tmp_x;
148:    fullf_j.x -= tmp_x;
     ...
251: } // for pairlist
```

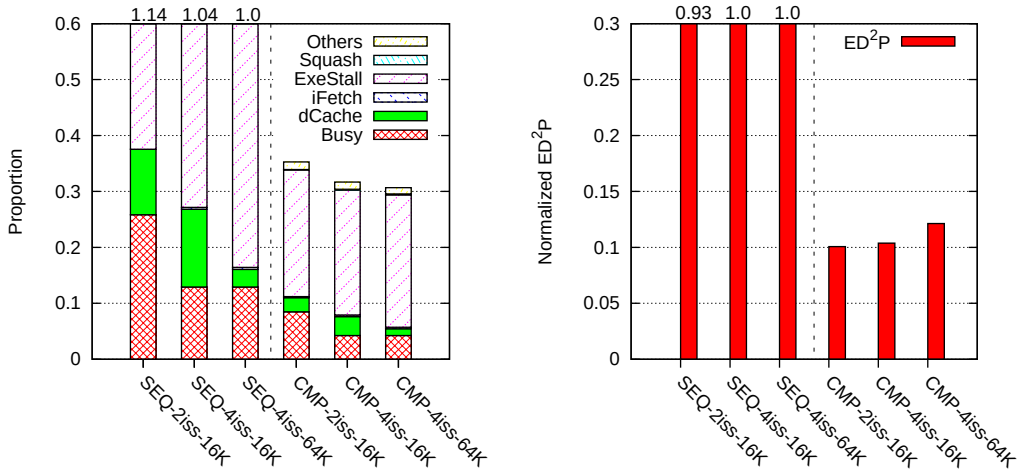
Figure 5.6: Source code snippets in NAMD

Thus we conclude that the heterogeneity we proposed is beneficial to energy efficiency not only for sequential applications, but also for their speculatively parallelized counterparts. Moreover, combined with the heterogeneous design, speculative parallelization could potentially bring energy efficiency to a level that cannot otherwise be attained.

5.4 Case Study

Many code segments exhibit interesting behaviors as they execute on different hardware configurations. In this section, we will focus on one such case and study these behaviors.

For benchmark NAMD, an important loop resides in source file ComputeNonboundedBased2.h, starting from line 12. Figure 5.4 shows some sample lines. Most array-based accesses are computed based on the a local variable `j` which is determined at the beginning of the loop by accessing array `pairlisti` using offset `k`, the iteration variable (line 13). In fact, most of these accesses are memory read operations and will not cause any dependence violation (line 69-71, and 142-148). The only memory accesses that



(a) Execution time breakdown (b) Energy efficiency

Figure 5.7: Execution time breakdown and energy efficiency of the NAMD code snippets running on different configurations. Note that in Figure 5.7(a) `Squash` and `Others` segments are always zero for sequential executions and the chopped portion belongs to `ExeStall`.

are susceptible to dependence violation are the reduction operations in line 97-99. Our TLS compiler synchronizes for these operations so that the transformed code is free of dependence violation caused by these reduction operations.

Figure 5.7(a) shows the execution time breakdown for executing this loop sequentially as well as in parallel on 4-core-CMP-based TLS systems. The cycle breakdown is exactly the same as described in Chapter 3.2. Bars represent different execution modes, core issue widths, and L1 cache capacities. In all configurations the L1 caches are 4-way set-associative and 32 bytes in block size. Figure 5.7(b) shows their energy efficiency measured in ED^2P. All the bars are normalized to SEQ-4iss-64K (sequential code running on a 4-issue core with 64K L1 cache), which is the sequential configuration that achieves the highest energy efficiency across all benchmarks.

There is virtually no speculation failure in all TLS configurations as expected. Furthermore, the cycles spent on cache accesses are quite small in both sequential and TLS

execution, and thus data cache performance is not a bottleneck for this loop. However, the execution stall (`ExeStall`) consumes a majority of the processor cycles, which indicates that individual iterations have low instruction-level parallelism.

Comparing the most aggressive sequential execution (SEQ-4iss-64K) and the least aggressive TLS execution (CMP-2iss-16K), we found that speculative parallelization reduces the execution time to 35.3%. When changing the 2-issue cores to 4-issue (CMP-2iss-16K vs. CMP-4iss-16K), the `Busy` segment is halved. However, the 4-issue cores pose more stress on the L1 cache, resulting in a 1/3 increase in the `dCache` segment. Nevertheless, higher issue width improves performance by a noticeable margin. Although it is sensible to increase the L1 cache size (CMP-4iss-16K vs. CMP-4iss-64K), the benefit of bigger caches is offset by the increased execution stall, due to the limited instruction-level parallelism available in each loop iteration. The result is only slightly better than that of the smaller cache size. Similar trend is observed when moving from SEQ-2iss-16K to SEQ-4iss-64K.

Higher issue width and larger cache capacity do not come free. When we trade off their performance benefit with the extra power consumption using the ED^2P metric, we found that configuration with 2-issue core and 16K L1 cache achieves the highest energy efficiency in both sequential and TLS execution modes. This is a strong case where simpler cores (2-issue) and smaller caches (16K) is significantly more efficient. Moreover, speculative parallelization further improves energy efficient by 90% compared to the best sequential configuration for this loop (CMP-2iss-16K vs. SEQ-2iss-16K).

These types of program behaviors are common when the SPEC benchmark suites are speculatively parallelized. Thus it is expected that heterogeneous design that matches the code behaviors can potentially improve energy efficiency by a significant margin.

5.5 Related Work

Among the large body of work [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] that support TLS on multi-threaded architectures, relatively little has addressed the energy efficiency issues of speculative execution. Packirisamy *et al* [62] compared and contrasted the energy efficiency of SMT and CMP in terms of supporting TLS workload under the same die area constraints. They examined how the interactions between speculative threads affect their energy efficiency in SMT-mode and CMP-mode execution. Renau *et al* [57] presented various energy-saving optimizations for a CMP-based TLS system. Our work explored different energy-saving opportunities. Namely, we achieved energy efficiency by allocating computation and caching components that match the performance characteristics of the running speculative threads. It is worth point out that the techniques proposed by Renau *et al*, such as energy-aware task pruning and elimination of low-return tasks, are orthogonal to our techniques and can potentially complement our system to further improve energy efficiency.

Heterogeneous systems with same-ISA heterogeneous cores are proposed by Kumar *et al* to improve energy efficiency for sequential programs [36] and multi-programming workloads [37]. They presented methodologies to model the various power consumption and the switch overheads, as well as policies for matching core sizes with application characteristics. Fair amount of energy was saved while only sacrificing a little in performance. In contrast, we proposed techniques to improve performance with minimal extra energy cost. In addition, we studied speculative threads that have unique sharing and interference patterns, managed these threads at a much finer granularity, and incorporated heterogeneous cache components, which is not considered by Kumar *et al*.

Suleman *et al* [38] proposed Asymmetric Chip Multiprocessor to speed up lock-based critical section execution in OpenMP applications. They presented a technique to leverage high-performance core(s) to accelerate the execution of critical sections. In contrast, our workloads do not have explicit lock-based critical sections and the system

deals resource allocation for *all* program segments. Moreover, we strive to balance performance and energy consumption, measured in energy-delay-squared product, which is not considered in their work.

Yang *et al* [39] proposed first-level cache designs that can be re-sized during program execution. Considerate energy-delay product reduction has been observed by having the ability of resizing both sets and associativities. Our work has shown that in the context of speculative threads that utilize cache ways as write buffers, resizing by sets is always a more energy-efficient choice.

5.6 Summary

In this chapter, we have explored the design space of a heterogeneous TLS system. With the help of an idealistic execution environment, we are able to efficiently experiment with a wide range of combinations of computing and caching components. With the following assumptions of (i) unlimited on-chip resources, (ii) an oracle thread allocator, and (iii) zero switching overhead, the heterogeneous system has a 39% upper-bound in energy efficiency improvement, compared to the best homogeneous TLS system. These assumptions will all be removed when we discuss the implementation details in Chapter 6.

We have also identified a subset of such components that can capture most of the potentials and is feasible to integrate. We have proposed an architecture design as a result and estimated that it could potentially improve energy efficiency by 37%, a close match to the 39% upper-bound. Our analysis also shows that each form of heterogeneity has contributed significantly towards improving energy efficiency. Thus the selections we have made are judicious.

Our main insight from this chapter is that the energy efficiency envelop could be pushed even higher with the combination of heterogeneous design and speculative parallelization.

Chapter 6

Heterogeneous System Implementation

In the previous chapter, we have shown promising margins of energy efficiency improvement by integrating heterogeneous components into a multi-core processor. To realize these potentials, two major hurdles must be overcome: switching overheads and resource allocation.

In this chapter, we will first present the runtime support for thread management. We will then identify various types of overheads that exist in the proposed heterogeneous system. Throttling mechanisms will be introduced to mitigate those overheads. We will also devise a competent resource allocation scheme that attempts to match program execution with the appropriate hardware configuration. Last, the entire heterogeneous system will be evaluated and contrasted with different design baselines.

6.1 Runtime Support

The runtime system maintains a hardware-based resource allocation table, similar to the decision table presented in Chapter 4.1. Indexed by the PC address of the first instruction of each program segment, the resource allocation table keeps track of thread

management decisions for each segment. The decision includes the preferred multi-threading types (for parallel segment only), core issue width, and L1 cache size.

Before a segment of execution starts, this table is queried: if no entry is found, the configuration of one 4-issue SMT core with 64K L1 cache is activated as the *default* mode. This is also the most energy-efficient homogeneous configuration on average. When a segment of execution completes in this configuration, the runtime system predicts the optimal configuration, and enter the decision in the resource allocation table.

6.2 Overhead Throttling Mechanisms

A realistic heterogeneous system must take into account various runtime overheads associated with thread migration and resource reconfiguration. In particular, when threads operate at a fine granularity, these overheads can be significant. In this section, we first identify these overheads and propose throttling mechanisms to mitigate the cost.

6.2.1 Types of runtime overheads

Thread migration and resource reconfiguration can incur performance penalty as well as extra energy consumption. In addition, an SMT core consumes more static power than a non-SMT core when executing a single thread.

Startup overhead

When runtime system migrates a thread to a core that is previously powered-off, execution cannot resume immediately. We model a 3000-cycle startup cost for the 4-issue core and an 800-cycle cost for the 2-issue core [43]. During this period, static power is being consumed.

Cache reconfiguration overhead

Powering-off a cache or part of a cache has additional costs. When a cache is powered off or re-sized to a smaller cache size (i.e., lower number of sets), dirty lines must be written back to the next-level cache, and some contents are discarded. When resizing to a bigger cache size (i.e., higher number of sets), the same operations must be performed, since some tag bits are shifted to the index bits after increasing the number of sets and the mapping of an address to a cache set is changed.

SMT overhead

Our SMT architecture is implemented by extending a superscalar core of the same issue width and replicating the renaming unit and the register file. When an SMT core is executing a single thread, the replicated structures are not used, but continue to consume static power. Thus the cores with SMT support can incur additional overhead in energy consumption.

6.2.2 Overhead throttling mechanisms

Frequent thread migration and resource reconfiguration are clearly detrimental. The central idea behind the various throttling heuristics is to reduce the frequency of such operations while still exploring the benefit of having heterogeneous components.

Coalescing requests for small segments

When a segment of execution is small, migrating threads or reconfiguring resources for the small segment probably would not justify the cost. In fact we have observed a number of situations where small segments with different configuration preferences interleave, creating tremendous overheads if reconfiguration is permitted whenever it is requested.

Ideally, we would prohibit thread migration and resource reconfiguration for all

segments that executes less than a certain number of cycles. But this would require knowing the cycle number before actually executing the invocation of a segment. Another problem is that although each invocation is small in cycle, a large number of consecutive invocations can together create a reasonable demand for thread migration or resource reconfiguration. Checking only the individual invocations would miss many opportunities.

To cope with these problems, we propose two heuristics: **accumulation** and **approximation**. The runtime system postpones thread migration and resource reconfiguration until the demand for such an operation has *accumulated* to a certain threshold. The accumulation is the cycle sum of the invocations of segments whose resource demands are the same but not met. The threshold should be related to the cost of the operation. Since cache reconfiguration exhibits additional costs, a higher threshold is set. Before the threshold is reached, reconfiguration requests are denied and the runtime system continues the execution with the current configuration. Meanwhile, the runtime system attempts to *approximate* the desired configuration with the components that are currently powered-on. For example, the current configuration is 4issue-16K and an incoming sequential segment demands 2issue-16K. If no 2-issue core is powered on in the system, the runtime system can schedule this segment on a configuration of a 4-issue core with 16K cache.

Postponing powering off components

Cores and caches should not be immediately powered off when they are no longer needed. Consider the case where small sequential segments interleave with large parallel segments. When a parallel segment completes, the runtime system will attempt to power-off the cores unused by the sequential segment. However, when the next long parallel segment starts, the cores must be powered on again. To avoid situations alike, powering off a component is always postponed for a short and deterministic duration in case that it will be needed in the near future.

Throttling for SMT overhead

As of the SMT overhead, we attempt to deal with it in a similar way. By power-gating the replicated register files, which are the source of most SMT overhead, the runtime system is able to significantly improve the energy efficiency of single-threaded execution on a SMT core. A 500-cycle delay is accounted when powering on the replicated register files and similar throttling mechanisms, such as accumulation, approximation and postponing powering-off, are applied.

6.3 Resource Allocation Schemes

The runtime support needs to decide, for each segment of execution, what resources should be allocated to execute the threads. Up to this point, an oracle makes this decision by first profiling each segment on all possible hardware configurations, and then choose the best one [36]. However, this is not realistic at runtime given that the number of possible configurations can be quite large, even with a few degrees of freedom.

In this section, we propose a realistic resource allocation scheme that utilizes hardware performance counters to monitor program execution and predict the most energy-efficient configuration for each segment. The scheme profiles each segment of execution once by allocating all threads to the *default* configuration that has one 4-issue SMT core with 64K L1 cache. Based on this profiling run, the optimal configuration is predicted and entered in the resource allocation table. A configuration contains the following information: core issue width (4-issue or 2-issue), L1 cache size (64K or 16K), and the multi-threading type (CMP or SMT, for parallel segment only).

6.3.1 Determining core issue width

The instruction per cycle (IPC) metric is often used to measure the performance of a processor; but it sometimes can be a poor indicator for whether the 4-issue core is efficiently utilized.

For program segments with a large number of long-latency operations, even though having low IPCs, they may still require a large reorder buffer (ROB) to hold the instructions and exploit instruction-level parallelism to achieve these IPCs. Normally a 4-issue core would have a larger ROB than the 2-issue core. Thus these program segments may be more energy efficient on the 4-issue core because of the extra ROB entries, even though the IPC statistics appear to be low.

We take the following approach to accommodate this effect. When profiling a segment on the 4-issue core, we record the percentage of instructions that are issued when they are farther from the ROB head than the size of the 2-issue core ROB. For example, in our design the 4-issue core has a ROB twice the size of that in the 2-issue core (Table 5.2). Thus the number of instructions that are issued from the second half of the ROB is counted. We denote the fraction of this count over the total number of issued instructions as *deep-issue rate*. Thus either a high deep-issue rate or a high IPC indicates that the segment can benefit from a more powerful core to exploit ILP; and thus the 4-issue core(s) should be selected; otherwise the 2-issue core(s) are selected.

6.3.2 Determining L1 cache size

The *reuse rate* of cache blocks is used as an indicator for whether a cache is efficiently utilized. The reuse rate is calculated as the fraction of the number of unique cache blocks that are accessed more than once over the total accesses to the cache over a fixed period of time. If the 64K cache has a high reuse rate during the profiling run, the larger cache(s) are selected; otherwise, the cache size will be reduced by turning off some of the sets.

6.3.3 Determining multi-threading type

Although SMT and CMP can both support speculative threads, they exhibit different power and performance characteristics. In CMP mode, the threads utilize multiple cores and this mode is a good match if thread-level parallelism is high. But if threads

often stall due to cache misses or synchronizations, resources in CMP mode cannot be efficiently utilized and the leakage power makes the situation even worse. In these circumstances, SMT is good alternative because it could hide some cache miss and synchronization latencies. Furthermore, it activates only one core and reduces leakage power. On the other hand, the contention between threads in SMT mode can reduce its energy efficiency; this situation can be worsen if threads compete for resources but are eventually squashed due to speculation failure.

To determine the optimal execution mode, we propose to measure the level of contention on the default 4-issue SMT core. If the threads are consistently stalled at the issue stage because the required resources or function units are occupied by other threads, CMP mode will be chosen so that the computation can be spread across multiple cores. Otherwise SMT mode is chosen. Eyerman *et al* [65] have proposed ways to obtain accurate cycle breakdown for threads executing under SMT mode, which can further improve the measurement of thread contentions.

New hardware counters must be integrated to the core and L1 cache to provide the required statistics. Note that we profile and make decisions only once for each segment. For instance, we start with the bigger cache and possibly convert it to a smaller cache; but the opposite transition never happens. It is the same with core size and execution mode. Potentially, we could periodically reset the resource allocation table so that all segments are profiled again to cope with phase change behaviors in the program.

6.4 System Evaluation

In this section, we will first evaluate the impact of thread management overheads as well as the effectiveness of throttling mechanisms. Then we will compare the oracle and the realistic resource allocation schemes. Last, the energy efficiency of our proposed system will be contrasted with different baselines.

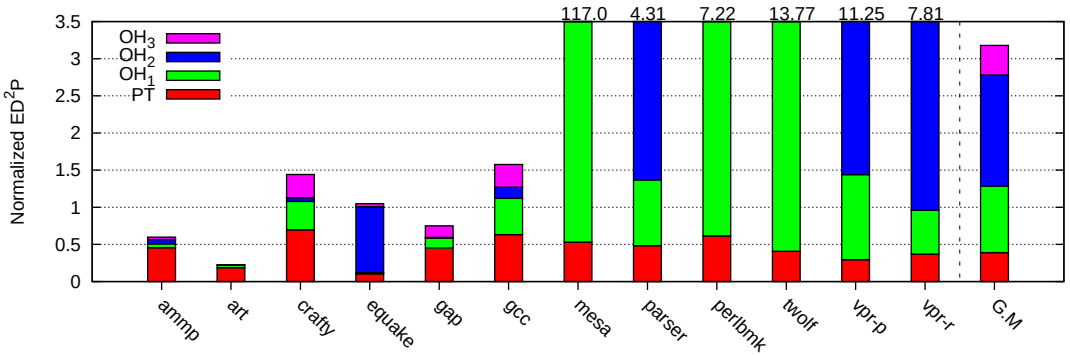
Table 6.1: Speculative thread characteristics in SPEC benchmarks.
Size: average number of dynamic instructions per thread.
Count: average number of threads per parallel segment.
Cov.: the fraction of time executing instructions in parallel segments.

CPU2000	Size	Count	Cov.	CPU2006	Size	Count	Cov.
ammp	363.2	23.7	99%	astar	276.6	799.5	71%
art	48.6	4323.4	100%	bzip2	162.7	7.0	46%
crafty	1764.2	1.9	13%	gobmk	919.9	3.3	20%
equake	113.9	6.5	93%	h264ref	175.5	23.6	81%
gap	654.9	101.2	27%	hmmer	221.6	148.2	96%
gcc	36.0	32.1	83%	lbm	407.4	122606.1	100%
mesa	103.7	2.5	60%	libquantum	17.3	795282.1	100%
parser	209.0	4.3	75%	mcf	62.6	16.6	97%
perlbmk	140.5	4.2	25%	milc	235.2	62186.9	85%
twolf	261.4	1.9	46%	namd	102.0	59.1	99%
vpr-p	219.7	4.9	56%	povray	746.3	2.4	64%
vpr-r	62.9	5.9	90%	sjeng	162.5	4.8	40%
Average	331.5	376.0	64%	sphinx3	302.2	32.4	98%
				Average	291.7	75474.8	77%

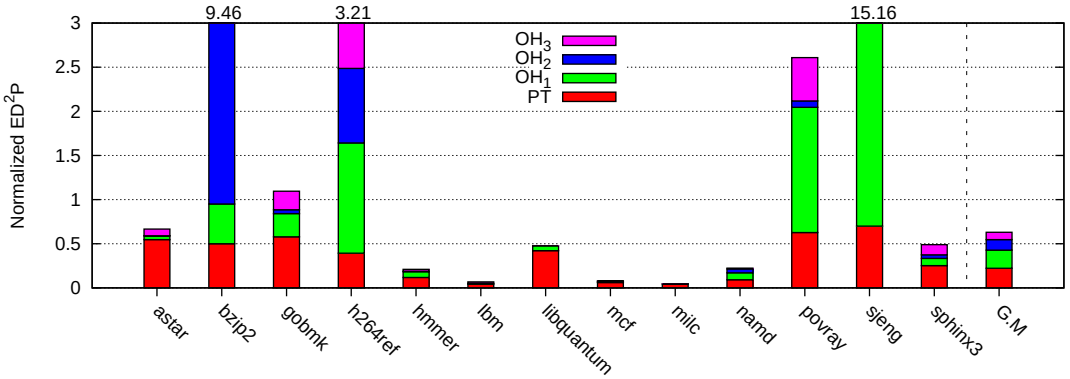
6.4.1 Benchmark characteristics

Table 6.1 summarizes some important characteristics of speculative threads for SPEC CPU2000 and CPU2006 benchmarks. We find that the average thread size, defined as the average number of dynamic instructions, is only about 330 for CPU2000 and 290 for CPU2006; and the average number of threads in a parallel segment is less than 150 in a majority of the benchmarks. Moreover, these speculative threads collectively cover 64% and 73% of total execution in CPU2000 and CPU2006, respectively.

Thus, to realize energy efficiency potentials discovered in the previous chapter, the runtime system must be able to manage threads at a *fine* granularity and cope with the associated overheads.



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 6.1: Measuring the three types of runtime overheads impacting the proposed heterogeneous system. Segments are described in the text. Note that `PT` is the last bar in Figure 5.4 and Figure 5.5.

6.4.2 Evaluation of runtime overheads

There are three types of runtime overheads in the proposed heterogeneous system, namely, startup overhead, cache reconfiguration overhead, and SMT overhead. We will measure their impacts separately.

Measuring cache reconfiguration overhead

In Figure 6.1, segment **PT** stands for the energy efficiency assuming no overhead. Segment $\mathbf{OH_1}$ shows the degradation in energy efficiency when the cache reconfiguration

overhead is first taken into account. This type of overhead is severely manifested in almost every benchmark, thus greatly degrading energy efficiency.

Take benchmark SJENG as an example. A speculatively parallelized loop, located at line 227 in source file neval.c, is most energy-efficient with a 16K-cache configuration. However, its neighboring sequential segment achieves optimal energy efficiency with a 64K-cache configuration. These two segments are nested in an outer loop that has a large iteration count. Thus cache resizing occurs frequently. Since cache contents are lost or partially lost during resizing, this results in severe performance degradation and increased energy consumption.

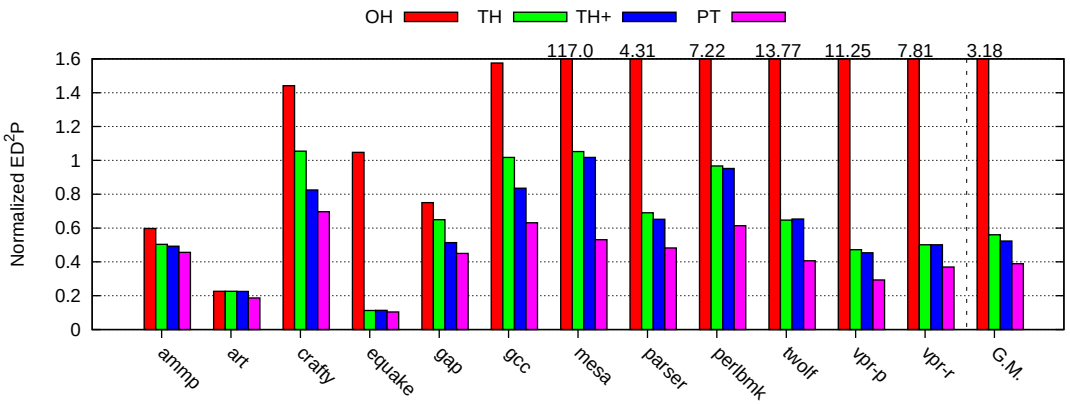
Overall, the heterogeneous system spends 47.9% more energy and suffers 22.9% performance loss, i.e. a 148.9% degradation in ED^2P due to cache reconfiguration overheads.

Measuring startup overhead

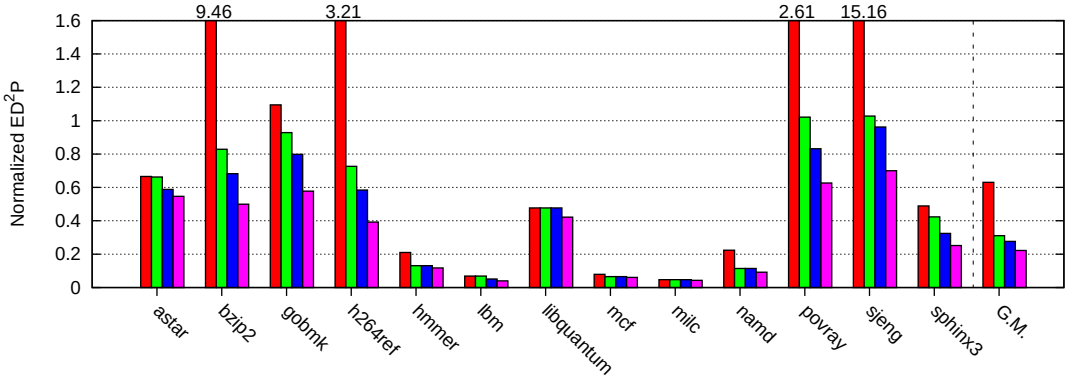
Segment $\mathbf{OH_2}$ corresponds to the additional degradation as the startup overhead is considered. A number of benchmarks, such as EQUAKE, MESA, PARSER, TWOLF, VPR-P, VPR-R, BZIP2, and H264REF, are greatly affected by the startup overhead. Overall, it leads to 160.7% additional degradation on top of $\mathtt{OH}_1$. Combining these two sources of overheads, energy efficiency is degraded by 309.6%.

Measuring SMT overhead

Segment $\mathbf{OH_3}$ stands for the additional degradation as the SMT overhead is further accounted. About half of the benchmarks, where the SMT core(s) are constantly used to run a single thread, suffer significantly: CRAFTY, GAP, GCC, MESA, PARSER, PERLBMK, TWOLF, VPR-P, VPR-R, BZIP2, GOBMK, H264REF, POVRAY, SJENG, and SPHINX3. Overall, it leads to another 61.5% degradation in energy efficiency on top of $\mathtt{OH}_2$. When the three types of overheads are combined, energy efficiency is degraded by as much as 370.7% compared to the heterogeneous system assuming none of these overheads.



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 6.2: Measuring the effectiveness of the throttling mechanisms. Bars are described in the text. Note that `OH` is $\texttt{OH}_3$ in Figure 6.1 and `PT` is the same as in Figure 6.1.

When compared with the homogeneous system, the 37.0% potential improvement (estimated in Chapter 5.3.2) in energy efficiency is turned into a 196.4% degradation.

6.4.3 Evaluation of throttling mechanisms

These runtime overheads clearly reverse the benefit of the proposed heterogeneous system. While they are inherent due to the heterogeneous nature of the system, we could mitigate their impacts by throttling the frequency of the causal operations.

In Figure 6.2, bar **OH** stands for the degradation caused by all three types of overheads. When we apply the throttling mechanisms for threads migration and cache reconfiguration (bar **TH**), the overheads are greatly mitigated for almost every benchmark.

Let's re-examine the case of the loop from SJENG, located at line 227 of source file neval.c. Although the neighboring sequential segment can potentially benefit from a larger cache, it is very small and will never reach the accumulation threshold. The throttling mechanisms correctly reject the cache reconfiguration request from each invocation of the small sequential segment and protect the cache from constantly being resized. Thus the overhead impact is significantly reduced.

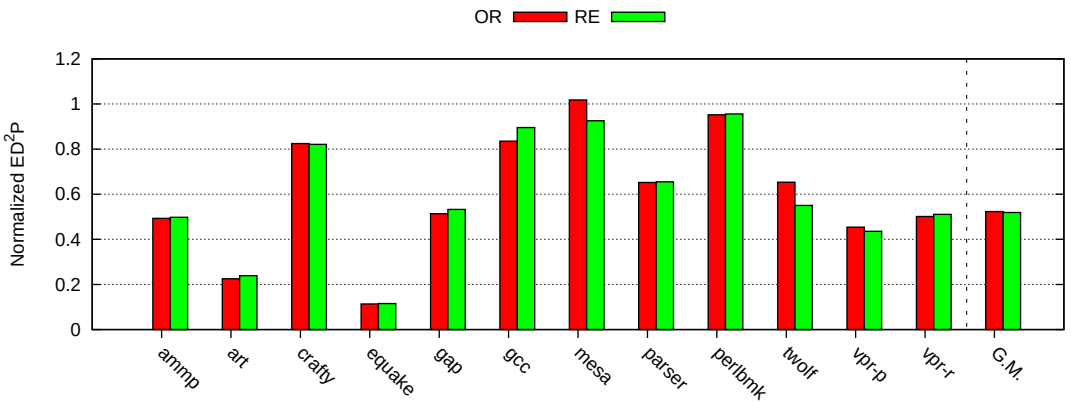
Further applying the mechanisms to power off SMT registers (bar **TH+**) has additional benefits in reducing the overhead impact. A number of benchmarks are able to take this advantage, such as CRAFTY, GAP, GCC, ASTAR, BZIP2, GOBMK, H264REF, POVRAY, SJENG, and SPHINX3.

Overall, these throttling mechanisms reduce the impact of overheads from 370.7% (`OH` vs. `PT`) to 29.0% (`TH+` vs. `PT`), a reduction as much as 92.2%. Though there may still be room for further improvement, we leave the investigations to our future work.

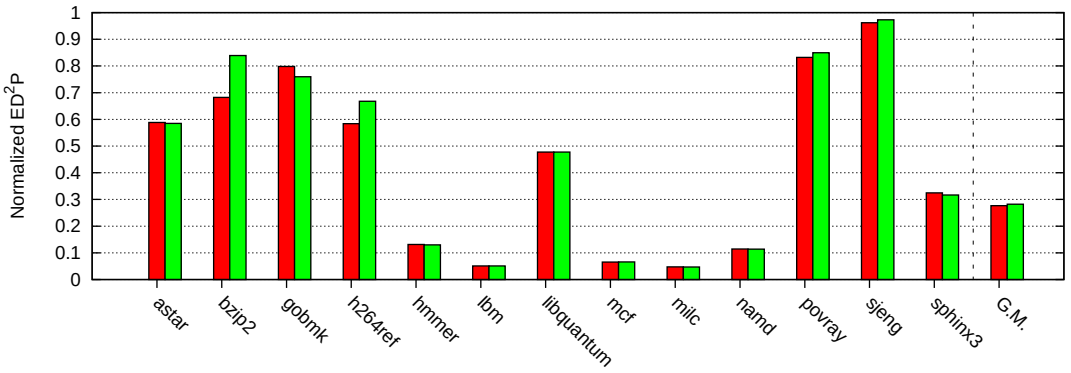
6.4.4 Evaluation of resource allocation schemes

The energy efficiency of the heterogeneous system using the realistic resource allocation schemes is shown as bar **RE** in Figure 6.3. It is able to realize most of the improvement demonstrated by the oracle allocation (bar **OR**). Only a few benchmarks exhibit noticeable degradation compared to the oracle allocation, such as GCC, BZIP2, H264REF and POVRAY, due to the deviations in configuration selections.

It is worth pointing out that the so-called oracle allocation is no longer optimal once the overheads are accounted. The cost of configuration change must be taken into consideration. In addition to the throttling mechanisms, the prediction-based realistic allocation could further reduce the number of configuration changes by enforcing one



(a) SPEC CPU2000 Benchmarks

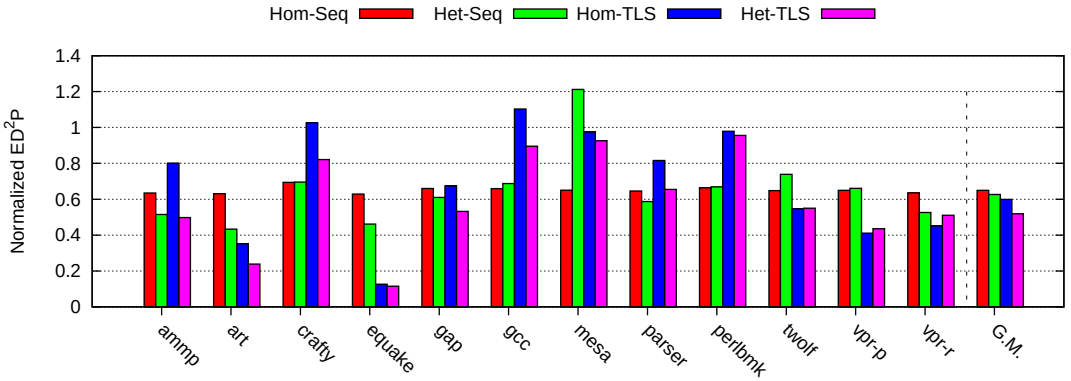


(b) SPEC CPU2006 Benchmarks

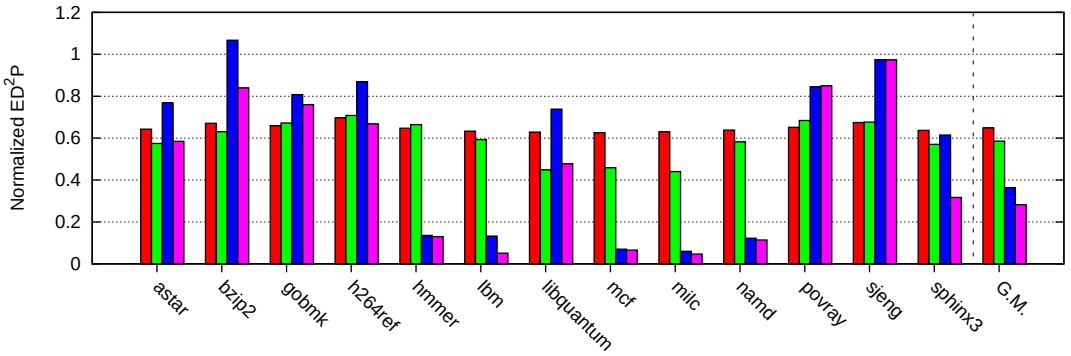
Figure 6.3: Comparing the energy efficiency of the oracle and the realistic resource allocation schemes. Bars are described in the text. Note that `OR` is `TH+` in Figure 6.2.

configuration per segment. This explains why `RE` is sometimes better than `OR`, for example in MESA, TWOLF, and GOBMK.

Overall, the difference between `OR` and `RE` is only 0.7%. Thus, we conclude that our proposed resource allocation schemes are making the right decisions in choosing the configuration for each segment of execution.



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 6.4: Comparing the energy efficiency of the proposed heterogeneous system with various baselines. Bars are described in the text. Note that `Hom-Seq` is `SEQ-hom` in Figure 5.5, `Hom-TLS` is `Hom-best` in Figure 5.4, and `Het-TLS` is `RE` in Figure 6.3.

6.4.5 Contrasting the heterogeneous system with various baselines

In this section we will contrast the energy efficiency of the proposed heterogeneous implementation (bar **Het-TLS**) with the following three baselines: (i) the sequential processor that achieves the highest energy efficiency among other sequential configurations (bar **Hom-Seq**), (ii) the heterogeneous sequential processor that has the same types of heterogeneity as in the proposed heterogeneous TLS implementation (bar **Het-Seq**), and (iii) the homogeneous TLS configuration that achieves the highest energy efficiency (bar **Hom-TLS**). Their energy efficiency corresponds to the bars in Figure 6.4.

Hom-Seq vs. Normalization Base

Our normalization base is the unmodified sequential code running on a 4-issue SMT core with 64K-4way L1 cache. The only disparity between `Hom-Seq` and the normalization base is whether the core has SMT support.

Since their execution time and dynamic power consumption are virtually the same, the difference between the `Hom-Seq` bar and 1 (normalization base) reveals the magnitude of the extra static power consumption when an SMT core is used to execute a single thread. Across all benchmarks, the degradation in energy efficiency due to SMT support is 54.1%.

Het-Seq vs. Hom-Seq

In Chapter 5.3.3 we estimated the upper-bound of energy efficiency improvement when heterogeneity is introduced to the sequential processor. In this section we have implemented such a heterogeneous sequential processor: a 4-issue core is augmented with a 2-issue core and they are connected through a switch to an L1 cache that can be resized between 64K-4way to 16K-4way by powering on and off some of the cache sets. This processor is also equipped with the same overhead throttling mechanisms.

Noticeable improvement in energy efficiency is observed in 14 benchmarks, such as ART and MILC. In other benchmarks, the improvement is limited by the overheads, even with throttling mechanisms. Across all benchmarks, the heterogeneous sequential processor is 6.8% better than the homogeneous sequential processor in terms of ED^2P.

Het-TLS vs. Het-Seq

Our findings about heterogeneous sequential processor are in line with [36]. Introducing speculative parallelism further magnifies the benefit of the heterogeneous design. This is demonstrated by the 37.5% improvement in ED^2P from a heterogeneous sequential processor (`Het-Seq`) to a heterogeneous multi-core processor that executes speculative

parallelized code (`Het-TLS`).

Conclusion: Speculative parallelism is key to improving energy efficiency for sequential applications.

Het-TLS vs. Hom-TLS

Our proposed heterogeneous TLS system outperforms the most energy-efficient homogeneous TLS configuration in 21 out of 25 benchmarks, and by a large margin in AMMP, ART, CRAFTY, GAP, GCC, PARSER, ASTAR, BZIP2, H264REF and SPHINX3. The only noticeable degradation is in VPR-P and VPR-R due to the overheads.

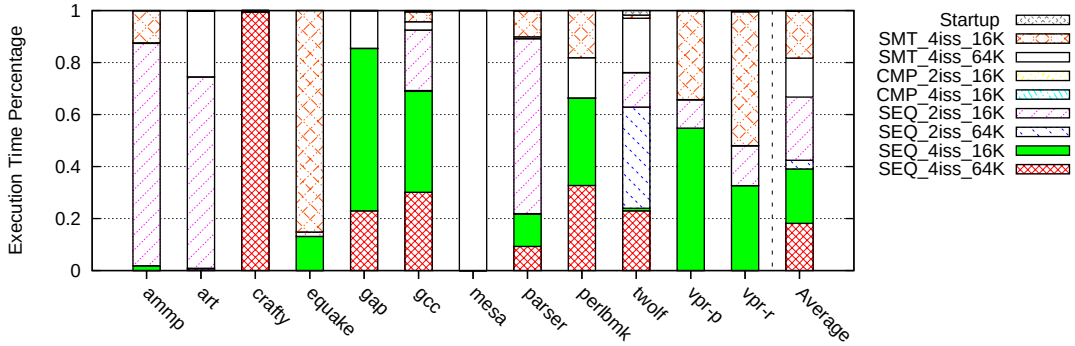
For SPEC CPU2000 benchmarks, `Het-TLS` leads to a 4.9% degradation in performance but reduces energy consumption by 21.7%, which is a 13.4% improvement in energy efficiency, measured by ED^2P. For SPEC CPU2006 benchmarks, `Het-TLS` improves performance by 4.5% and saves energy by 15.3%, which is a 22.4% improvement in energy efficiency. Across all benchmarks, the energy efficiency is improved by 18.2% compared to the homogeneous TLS baseline.

Conclusion: The heterogeneous design is essential to further boost energy efficiency over prior TLS implementations by a significant portion.

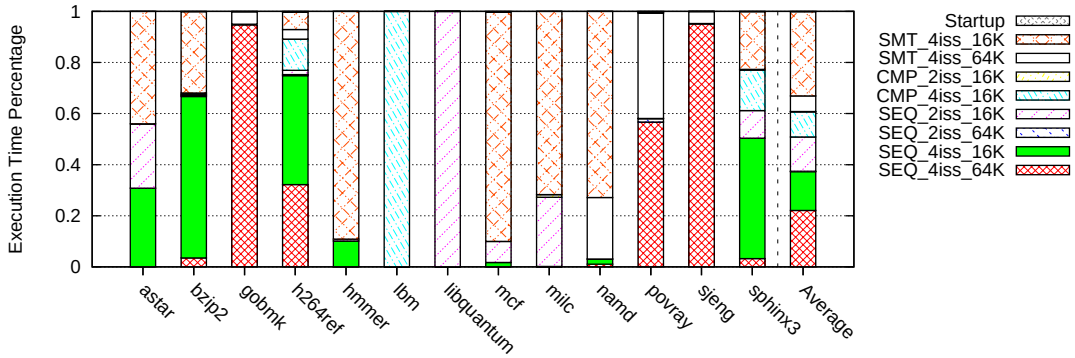
Het-TLS vs. Hom-Seq

Last, the proposed heterogeneous TLS system is compared with the most energy-efficient sequential configuration. For a few benchmarks that have large sequential portions with high instruction-level parallelism, `Het-TLS` is less efficient than `Hom-Seq`, such as in CRAFTY, GCC, POVRAY, SJENG, and etc. Nevertheless, `Het-TLS` is significantly better than `Hom-Seq` in 15 out of 25 benchmarks.

For SPEC CPU2000 benchmarks, `Het-TLS` increase energy consumption by 1.1% but improves performance by 12.5%, which is a 20.2% improvement in energy efficiency, measured by ED^2P. For SPEC CPU2006 benchmarks, `Het-TLS` improves performance by 45.5% at the cost of consuming 7.9% more energy, which is a 56.5% improvement



(a) SPEC CPU2000 Benchmarks



(b) SPEC CPU2006 Benchmarks

Figure 6.5: Execution time breakdown of the programs running on the proposed heterogeneous system. Stack segments correspond to the percentage of execution time a certain hardware configuration is activated or the stalled cycles due to powering on a component (segment `Startup`).

in energy efficiency. Across all benchmarks, the energy efficiency is improved by 41.8% compared to the homogeneous sequential baseline.

Conclusion: Through the heterogeneous design, performance can be improved over sequential execution in an energy-efficient way.

6.4.6 Measurement of configuration usage breakdown

We have measured, in the proposed heterogeneous system, the fraction of execution time in which each configuration is activated. Figure 6.5 reveals diverse use of configurations.

In SPEC CPU2000 benchmarks, CMP-based configurations are rarely used due to their low level of thread-level parallelism. In SPEC CPU2006 benchmarks, CMP-based configurations are activated for 10% of total execution time, because the higher level of thread-level parallelism available could justify the cost of using CMP. Across all the benchmarks, the 4-issue and 2-issue core(s) are used for 79% and 21% of total execution time, respectively; and the 64K-4way and 16K-4way cache(s) are used for 32% and 67% of total execution time, respectively.

Furthermore, thread migration and/or cache resizing have taken place in the same program at runtime. For example, in ASTAR, both 4-issue-core and 2-issue-core configurations are used significantly; this is an evidence for thread migration. In H264REF, both 64K and 16K cache(s) are activated for a large percentage; this is an evidence for cache resizing. Similar cases can be found in a number of benchmarks.

It is also worth pointing out that the cycles stalled due to powering on a component are minimal (0.1%). Thus the throttling mechanisms are very effective in reducing the startup overheads.

6.5 Related Work

A large number of proposals have discussed dynamic tuning for improving energy efficiency of microprocessors. Wu *et al* [40, 41] presented the runtime optimizer framework that uses dynamic voltage and frequency scaling (DVFS) to manage energy/performance tradeoffs. This framework is based on dynamic instrumentation: initial test instructions are inserted to hot code regions to collect execution profile; then mode set/reset instructions are inserted at the entry/exit of candidate code segments. Isci *et al* [42, 43] collected control flow, performance counters and live power measurement information from running applications, and found that performance-counter-based method consistently provided better representation of power behaviors. Bhattacharjee *et al* [44] proposed thread criticality predictor and then applied dynamic optimization and DVFS based on

the predicted criticality of the threads. While DVFS is not evaluated in this thesis, it is orthogonal to our approach and can be adopted to further improve energy efficiency of TLS execution.

As more and more transistors are integrated onto a single die, efficiently utilizing on-chip resource under a certain workload becomes increasingly challenging. The work from [66, 67, 68, 69, 70] studied thread migration for better resource utilization. However, their proposed schemes worked at coarse granularity and may not support speculative threads. Our work focused on speculative threads and tackled fine-grained thread management overheads.

Furthermore, the energy-efficient support for TLS could be implemented on alternative heterogeneous architectures, such as the Wisconsin Forwardflow architecture [71]. In this architecture, the execution logic can either scale up to run a single thread or scale down to run multiple threads. We could dynamically adjust the number of hardware threads based on the availability of instruction level and thread level parallelism exhibited in the running segment of execution.

6.6 Summary

Following the investigation of Chapter 5, we have addressed two major issues in implementing a heterogeneous multi-core system for executing speculative threads.

First, we identify several runtime switching overheads and propose throttling mechanisms to mitigate their impacts by reducing the frequency of the causal operations. Our evaluation shows that the throttling mechanisms can effectively eliminate 92% of such overheads.

Second, realistic resource allocation schemes are proposed to dynamically monitor thread characteristics and match with them the most appropriate hardware configuration. We adopt a divide-and-conquer strategy that makes decision on multi-threading type, core issue width and cache size separately. Our evaluation shows that the realistic

resource allocation schemes achieve energy efficiency on par with the oracle allocator. Thus we conclude the realistic allocator makes the right decisions.

When contrasting the proposed heterogeneous system with various design baselines, we find that the heterogeneous system has harnessed a large portion of the energy efficiency improvement upper-bound estimated in the previous chapter. It achieves 18% higher energy efficiency compared to the best homogeneous TLS implementation. When compared to sequential execution, the heterogeneous system achieves 29% better performance at the cost of 4% more energy consumption. Thus we conclude that the performance is improved in an energy-efficient way.

Chapter 7

Conclusion and Future Work

With the emergence of multi-core processors, various aggressive execution models have been proposed to exploit fine-grained thread-level parallelism, taking advantage of the fast on-chip interconnection communication. However, the aggressive nature of these execution models often leads to unpredictable performance and/or excessive energy consumption incommensurate to the execution time reduction. Under the context of thread-level speculation (TLS), this dissertation proposed dynamic optimizations to improve the performance of speculative threads and their energy efficiency.

First, we presented methodologies to accurately evaluate speculative thread performance at runtime. Our approach is to predict the sequential execution time from the parallel execution and deem their difference in cycle as a metric to quantitatively measure the efficiency of speculative execution. With two important adjustments, execution stall scaling and cache behavior classification, we are able to make correct predictions for 94% of the loops and approximate the actual sequential execution time with an accuracy of 86%.

Our TLS infrastructure, like many others, focuses on parallelizing loops from sequential programs. Many such programs contain multiple nested loops, and thus the

optimization system is required not only to identify and parallelize loops that can benefit from TLS, but also to select the right level of loop to maximize the overall performance gain. To achieve this purpose, we designed an execution framework that facilitates the dynamic monitoring, evaluation, and adjustment of speculative threads, where hardware-based performance counters are programmed to collect necessary runtime information. This dissertation also proposed performance tuning policies that search for the desired loop levels for spawning speculative threads in order to maximize performance. Compared to the previous system that uses the state-of-the-art static analysis, our dynamic tuning system further improves performance by 5.1% on average over a large set of programs from the SPEC CPU2000 and CPU2006 benchmark suites. Compared to the original sequential execution, our dynamic tuning system has an average speedup of 1.38x.

By examining the design space of a heterogeneous multi-core system with TLS support, this dissertation identifies opportunities for energy efficiency improvement. By building an idealistic execution environment and making a number of simplifying assumptions, we are able to efficiently experiment multi-core configurations with a diverse combinations of computing and caching components. Our finding is that the heterogeneous system has a 39% upper-bound in energy efficiency improvement compared to the best homogeneous TLS system. We also identified a subset of such components that can capture most of the potentials and is feasible to integrate. An architecture design is proposed as a result and it is estimated to improve energy efficiency by 37%, a close match to the 39% upper-bound. Moreover, our analysis also shows that each form of heterogeneity has contributed significantly towards improving energy efficiency. Thus the selections we have made are judicious.

To implement such a heterogeneous system, two major issues must be addressed. First, we identified several runtime switching overheads and proposed throttling mechanisms to mitigate their impacts by reducing the frequency of the causal operations.

Our evaluation showed that the throttling mechanisms can effectively eliminate 92% of such overheads. Second, realistic resource allocation schemes were proposed to dynamically monitor thread characteristics and match with them the most appropriate hardware configuration. Our evaluation shows that the realistic resource allocation schemes achieve energy efficiency on par with the oracle allocator. With all simplifying assumptions removed, the proposed heterogeneous system achieves 18% higher energy efficiency compared to the best homogeneous implementation. When compared to sequential execution, the heterogeneous system achieves 29% better performance at the cost of 4% more energy consumption. Thus we conclude that the performance is improved in an energy-efficient way.

The key insights of this dissertation are summarized as the following:

1. Being able to accurately understand cache behaviors and the effects on execution stall is necessary for determining the performance impact of speculative threads.

2. Quantitative evaluation of speculative thread efficiency and incorporation of static analysis are two key ingredients to a successful dynamic tuning policy that maximizes performance gain.

3. The energy efficiency envelop could be pushed higher with the combination of heterogeneous design and speculative parallelization.

4. On-chip heterogeneity and dynamic resource allocation are two key elements for achieving performance improvement in an energy-efficient way.

7.1 Future Work

The research presented in this dissertation could be extended in the following areas:

7.1.1 Dynamic optimization of other assistant threads

Numerous execution models involving the creation of *assistant threads* have been proposed to utilize the emerging multicore to satisfy diverse performance or non-performance requirements. For many such threads, their execution often introduces significant performance variations and resource competition. Thus, judicious utilization of these threads is key to application performance.

One such type of assistant threads is helper thread [72, 73, 74, 75, 76, 27, 29]. A helper thread improves the performance of an application by bringing data into the shared cache before they are needed. However, if deployed improperly, prefetching threads can also degrade application performance by polluting the cache or saturating shared resources, such as the off-chip pin bandwidth or the bus, creating performance bottleneck. We can apply the following steps to dynamically optimize the performance of a helper thread: (i) executing the program with a helper thread and configuring the hardware performance monitors to dynamically collect a performance profile that contains information regarding how prefetched lines are used by the main thread, whether prefetched data displaces useful data, and whether the helper thread contributes to reduced data cache stalls; (ii) isolating the performance impact of the helper thread using the dynamically collected performance profile, estimating the performance of the main thread in the absence of the helper thread using the profile information; and (iii) enabling/disabling or re-optimizing the helper-thread based on its performance impact.

7.1.2 Adaptation to phase changes

Programs exhibit phase behaviors. During different phases of execution, speculative threads may behave differently. Thus, the dynamic optimization system must adapt to these changes and potentially re-optimize speculative threads accordingly. In this dissertation, a simple mechanism is used to adapt to phase changes (Chapter 4.2 and Chapter 6.3). The performance and decision tables are reset periodically, and the optimization

decisions are re-evaluated. This mechanism can potentially introduce unnecessary overhead if the performance characteristics of the speculative threads remain unchanged. Many researchers have been working on detecting phase changes. Our system can adopt such phase change detection mechanisms so that optimization decisions are re-evaluated only when a phase change is observed.

7.1.3 Improving resource allocation

This dissertation proposes a resource allocator that makes decisions with simple policies (Chapter 6.3). Generally it works well, however with a few exceptions. This allocator only examines a few performance characteristics of speculative threads, but does not consider energy consumption. On the other hand, modern processors that have on-chip voltage/current meters can provide accurate power consumption measurement. Therefore, a comprehensive approach that accounts both performance and energy consumption measurements can potentially improve such decision making.

7.1.4 Collaborative thread execution environment

Future computer systems are likely to consist of multiple cores and many different types of threads. Such systems can either be homogeneous or heterogeneous and the threads can come from the same application or from multiple applications. Ideally, these threads should execute collaboratively and share system resources. How to best allocate resources and schedule these threads presents a challenge to the operating system. We believe that the experience and insights we have learned from optimizing speculative threads serve as an important step in optimizing the multi-programming workloads and eventually creating an environment where threads execute collaboratively.

References

[1] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[2] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 226–236, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[3] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 58–67, New York, NY, USA, 1992. ACM.

[4] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 43–, Washington, DC, USA, 2002. IEEE Computer Society.

[5] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG10.3 working conference*

on Parallel architectures and compilation techniques, PACT '95, pages 109–121, Manchester, UK, 1995. IFIP Working Group on Algol.

[6] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 414–425, New York, NY, USA, 1995. ACM.

[7] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 58–69, New York, NY, USA, 1998. ACM.

[9] Pedro Marcuello and Antonio González. Clustered speculative multithreaded processors. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 365–372, New York, NY, USA, 1999. ACM.

[10] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, pages 303–, Washington, DC, USA, 1999. IEEE Computer Society.

[11] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[12] Jenn-Yaun Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Trans. Comput.*, 48:881–902, September 1999.

[13] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23:253–300, August 2005.

[14] Venkatesan Packirisamy, Shengyue Wang, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. Supporting speculative multithreading on simultaneous multithreaded processors. In *Proceedings of the 13th International Conference on High Performance Computing*, 2006.

[15] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 81–92, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[16] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 55–, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 71–81, New York, NY, USA, 2004. ACM.

[18] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN*

2004 conference on Programming language design and implementation, PLDI '04, pages 59–70, New York, NY, USA, 2004. ACM.

[19] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 269–279, New York, NY, USA, 2005. ACM.

[20] Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula, Antonia Zhai, and Pen chung Yew. Loop selection for thread-level speculation. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[21] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 158–167, New York, NY, USA, 2006. ACM.

[22] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 205–214, New York, NY, USA, 2007. ACM.

[23] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 14–, Washington, DC, USA, 1998. IEEE Computer Society.

[24] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 77–84, New York, NY, USA, 1998. ACM.

[25] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[26] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 180–, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. ACM.

[29] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.

[31] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[32] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 277–286, New York, NY, USA, 2008. ACM.

[33] Mihai Burcea. stOMP: A specializing thread library for OpenMP. Master's thesis, University of Toronto, 2005.

[34] Jae W. Lee, Man Cheuk Ng, and Krste Asanovic. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 89–100, Washington, DC, USA, 2008. IEEE Computer Society.

[35] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[37] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st annual international*

symposium on Computer architecture, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.

[38] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 253–264, New York, NY, USA, 2009. ACM.

[39] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 151–, Washington, DC, USA, 2002. IEEE Computer Society.

[40] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 178–189, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Canturk Isci and Margaret Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Washington, DC, USA, 2006. IEEE Computer Society.

[43] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 290–301, New York, NY, USA, 2009. ACM.

[45] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst.*, 15:322–354, August 1997.

[46] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 191–202, New York, NY, USA, 1996. ACM.

[47] Open64 Developers. Open64 compiler and tools. http://www.open64.net, 2001.

[48] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 171–183, New York, NY, USA, 2002. ACM.

[49] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the international symposium on Code generation*

and optimization: feedback-directed and runtime optimization, CGO '04, pages 39–, Washington, DC, USA, 2004. IEEE Computer Society.

[50] Shengyue Wang. *Compiler Techniques for Thread-Level Speculation*. PhD thesis, University of Minnesota - Twin Cities, 2007.

[51] SimpleScalar LLC. The SimpleScalar tool set. http://www.simplescalar.com/, 2004.

[52] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[53] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. In *In Technical Report of Compaq Computer Corporation*. 2001.

[54] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: a power-performance simulator for interconnection networks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 294–305, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[55] Erez Perelman, Marzia Polito, Jean-Yves Bouguet, John Sampson, Brad Calder, and Carole Dulong. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 88–88, Washington, DC, USA, 2006. IEEE Computer Society.

[56] Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society.

[57] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. Thread-level speculation on a CMP can be energy efficient. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 219–228, New York, NY, USA, 2005. ACM.

[58] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 179–188, New York, NY, USA, 2005. ACM.

[59] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 175–184, New York, NY, USA, 2006. ACM.

[60] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[61] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 3–14, Washington, DC, USA, 2010. IEEE Computer Society.

[62] V. Packirisamy, Yangchun Luo, Wei-Lung Hung, A. Zhai, Pen-Chung Yew, and Tin-Fook Ngai. Efficiency of thread-level speculation in SMT and CMP architectures - performance, power and thermal perspective. In *Proceedings of the International Conference on Computer Design*, 2008.

[63] Yiannakis Sazeides, Rakesh Kumar, Dean M. Tullsen, and Theofanis Constantinou. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Comput. Archit. Lett.*, 4:1–, January 2005.

[64] Hashem Hashemi Najaf-abadi, Niket Kumar Choudhary, and Eric Rotenberg. Core-selectability in chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 113–122, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in SMT processors. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 133–144, New York, NY, USA, 2009. ACM.

[66] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 124–133, New York, NY, USA, 2006. ACM.

[67] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM.

[68] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 283–292, New York, NY, USA, 2006. ACM.

[69] J.Chen and L.K. John. Energy-aware application scheduling on a heterogeneous multi-core system. In *International Symposium on Workload Characterization*, 2008.

[70] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.

[71] Dan Gibson and David A. Wood. Forwardflow: a scalable core for power-constrained CMPs. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 14–25, New York, NY, USA, 2010. ACM.

[72] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 159–170, New York, NY, USA, 2002. ACM.

[73] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[74] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 40–51, New York, NY, USA, 2001. ACM.

[75] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 269–280, New York, NY, USA, 2000. ACM.

[76] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 257–268, New York, NY, USA, 2000. ACM.

Appendix A

Glossary and Acronyms

Care has been taken in this thesis to minimize the use of jargon and acronyms, but this cannot always be achieved. This appendix defines jargon terms in a glossary, and contains a table of acronyms.

A.1 Glossary

- **Premature serialization** – If the efficiency of speculative threads for a loop is not evaluated quantitatively, this loop could be serialized prematurely, even though overall parallel execution still improve performance over sequential execution.
- **Parallel code overhead** – If a loop is instrumented for parallelization but eventually executed sequentially, the extra instructions incur execution overheads at runtime.
- **Parallel segment** – The sequence of dynamic instructions that is speculatively parallelized.
- **Sequential segment** – The sequence of dynamic instructions executed between two parallel segments.

- **Oracle resource allocator** – A unrealistic resource allocator that can determine the most energy-efficient hardware configuration for each parallel and sequential segment.

- **Processing block** – A block of components that includes an SMT core, a non-SMT core, and a resizable cache. They are connected through a two-way switch.

- **Deep issue rate** – The percentage of instructions issued from the back of the reorder buffer on an out-of-order processor.

- **Reuse rate** – The percentage of unique cache blocks that are accessed more than once over a period of time.

A.2 Acronyms

Table A.1: Acronyms

Acronym	Meaning
TLS	Thread-Level Speculation
TM	Transactional Memory
SMT	Simultaneous Multi-Threading
CMP	Chip Multi-Processing
ISA	Instruction-Set Architecture
ILP	Instruction-Level Parallelism
TLP	Thread-Level Parallelism
EDP	Energy-Delayed Product
ED^2P	Energy-Delayed-Squared Product
ROB	Re-Order Buffer
L1	First-Level (cache)

Continued on next page

Table A.1 – continued from previous page

Acronym	Meaning
L1D	First-Level Data (cache)
L2	Second-Level (cache)
CAM	Content-Addressable Memory
PC	Program Counter
IPC	Instruction Per Cycle
NOP	No Operation
DVFS	Dynamic Voltage and Frequency Scaling

Appendix B

Loops Parallelized

The proposed dynamic performance tuning policy often identifies a different set of loops for parallelization and results in higher performance compared to the static thread management. In this chapter we contrast the loops selected by the static and dynamic approaches for each studied benchmark. The tuning policy used in the dynamic approach is the `Quant+StaticHint` policy (Chapter 4.2.5).

We report the following information for loops that have significant coverage in each benchmark:

- Source file name,
- Starting line number,
- Loop coverage, and
- Loop speedup over sequential execution.[1]

Our observation is that for SPEC CPU2000 benchmark suite, loops are selected quite differently between static and dynamic approaches. Whereas for SPEC CPU2006 benchmark suite, static and dynamic approaches are in agreement in most cases.

[1] Speedup is normalized to the sequential execution of the respective parallelized code, to discount the impact of parallel code overhead (Chapter 4.3.3)

B.1 SPEC CPU2000 Benchmarks

Table B.1: Loops parallelized for AMMP

Static Loops	Coverage	Speedup
rectmm.c:562	80%	1.26
vnonbon.c:482	9%	1.18
Dynamic Loops	Coverage	Speedup
rectmm.c:995	29%	1.98
vnonbon.c:482	9%	1.18

Table B.2: Loops parallelized for ART

Static Loops	Coverage	Speedup
scanner.c:611	39%	1.1
scanner.c:584	17%	2.88
Dynamic Loops	Coverage	Speedup
scanner.c:615	39%	3.08
scanner.c:584	18%	2.74

Table B.3: Loops parallelized for EQUAKE

Static Loops	Coverage	Speedup
quake.c:463	20%	1.39
quake.c:1204	48%	2.35
quake.c:475	9%	3.89
Dynamic Loops	Coverage	Speedup
quake.c:463	21%	1.39
quake.c:1204	48%	2.35
quake.c:475	9%	3.89
quake.c:470	11%	3.91
quake.c:458	4%	2.83

Table B.4: Loops parallelized for GAP

Static Loops	Coverage	Speedup
statemen.c:353	9%	1.1
gasman.c:772	16%	2.3
Dynamic Loops	Coverage	Speedup
integer.c:743	10%	1.96
integer.c:350	5%	1.82
ProdInt:803	11%	2.0
gasman.c:772	14%	2.3

Table B.5: Loops parallelized for GCC

Static Loops	Coverage	Speedup
sched.c:3831	6%	1.96
regclass.c:717	25%	2.02
flow.c:1091	12%	0.57
flow.c:1422	25%	2.16
Dynamic Loops	Coverage	Speedup
sched.c:3831	4%	1.9
flow.c:1634	11%	2.37
flow.c:1422	16%	2.15

Table B.6: Loops parallelized for GZIP

Static Loops	Coverage	Speedup
deflate.c:590	17%	0.9
inflate.c:513	15%	1.09
deflate.c:678	67%	1.1
Dynamic Loops	Coverage	Speedup
deflate.c:761	9%	1.02
deflate.c:729	5%	1.35
deflate.c:479	49%	1.8
deflate.c:653	5%	1.02
inflate.c:547	7%	1.0

Table B.7: Loops parallelized for MCF

Static Loops	Coverage	Speedup
implicit.c:228	34%	1.28
mcfutil.c:80	54%	2.84
pbeampp.c:181	6%	2.42
Dynamic Loops	Coverage	Speedup
mcfutil.c:82	53%	2.19
implicit.c:246	34%	2.26

Table B.8: Loops parallelized for MESA

Static Loops	Coverage	Speedup
triangle.c:730	13%	0.99
Dynamic Loops	Coverage	Speedup
shade.c:366	5%	2.61
vbrender.c:897	87%	1.81

Table B.9: Loops parallelized for PARSER

Static Loops	Coverage	Speedup
main.c:1605	13%	0.98
parse.c:316	11%	1.0
build-disjuncts.c:112	5%	2.13
parser.c:582	17%	1.01
Dynamic Loops	Coverage	Speedup
parse.c:511	46%	1.62

Table B.10: Loops parallelized for PERLBMK

Static Loops	Coverage	Speedup
util.c:1559	2%	0.69
regexec.c:1306	15%	1.12
Dynamic Loops	Coverage	Speedup
regexec.c:1493	30%	1.01
util.c:1559	4%	1.47
regexec.c:637	9%	1.15

Table B.11: Loops parallelized for TWOLF

Static Loops	Coverage	Speedup
dimbox.c:82	7%	1.2
dimbox.c:211	13%	2.65
dimbox.c:254	11%	2.57
Dynamic Loops	Coverage	Speedup
dimbox.c:82	7%	1.18
dimbox.c:211	13%	2.51
dimbox.c:254	11%	2.56

Table B.12: Loops parallelized for VORTEX

Static Loops	Coverage	Speedup
sa.c:760	39%	0.99
oa1.c:428	18%	1.08
bmtobj.c:828 (or 765)	15%	1.18
bmt01.c:468	4%	0.9
bmt01.c:82	7%	0.99
Dynamic Loops	Coverage	Speedup
sa.c:760	39%	1.03
oa1.c:428	18%	1.31
bmtobj.c:828 (or 765)	15%	1.17
bmt01.c:468	4%	1.0

Table B.13: Loops parallelized for VPR-PLACE

Static Loops	Coverage	Speedup
place.c:897	53%	2.34
place.c:947	7%	2.14
Dynamic Loops	Coverage	Speedup
place.c:897	52%	2.27
place.c:947	7%	2.14
place.c 1100	9%	1.66
place.c 1079	9%	1.66

Table B.14: Loops parallelized for VPR-ROUTE

Static Loops	Coverage	Speedup
route.c:777	53%	1.73
route.c:1081	35%	1.01
Dynamic Loops	Coverage	Speedup
route.c:777	53%	1.74

B.2 SPEC CPU2006 Benchmarks

Table B.15: Loops parallelized for ASTAR

Static Loops	Coverage	Speedup
Way2_.cpp:100	41%	1.2
Way_.cpp:57	23%	0.87
RegWay_.cpp:39	25%	1.38
Dynamic Loops	Coverage	Speedup
Way2_.cpp:100	42%	1.1
RegWay_.cpp:35	30%	2.47
Way_.cpp:228	23%	1.01

Table B.16: Loops parallelized for BZIP2

Static Loops	Coverage	Speedup
compress.c:207	32%	1.28
Dynamic Loops	Coverage	Speedup
compress.c:207	36%	1.27

Table B.17: Loops parallelized for GOBMK

Static Loops	Coverage	Speedup
engine/worm.c:880	13%	1.0
engine/matchpat.c:417	8%	1.0
Dynamic Loops	Coverage	Speedup
engine/reading.c:4171	16%	1.4

Table B.18: Loops parallelized for H264REF

Static Loops	Coverage	Speedup
mv-search.c:982	25%	1.26
mv-search.c:1448	8%	2.23
mv-search.c:394	28%	3.51
mv-search.c:1348	9%	0.98
Dynamic Loops	Coverage	Speedup
mv-search.c:982	21%	1.26
mv-search.c:1448	7%	2.33
mv-search.c:394	37%	4.34
mv-search.c:1367	8%	1.47

Table B.19: Loops parallelized for HMMER

Static Loops	Coverage	Speedup
fast_algorithms.c:133	94%	2.0
Dynamic Loops	Coverage	Speedup
fast_algorithms.c:133	94%	2.0

Table B.20: Loops parallelized for LBM

Static Loops	Coverage	Speedup
lbm.c:186	100%	3.59
Dynamic Loops	Coverage	Speedup
lbm.c:186	100%	3.59

Table B.21: Loops parallelized for LIBQUANTUM

Static Loops	Coverage	Speedup
gates.c:170	28%	1.09
gates.c:89	66%	1.13
gates.c:61	5%	1.12
Dynamic Loops	Coverage	Speedup
gates.c:170	28%	1.09
gates.c:89	66%	1.13
gates.c:61	5%	1.12

Table B.22: Loops parallelized for MILC

Static Loops	Coverage	Speedup
quark_stuff.c:1420	7%	3.74
quark_stuff.c:1396	5%	3.95
quark_stuff.c:1375 146	14%	3.91
quark_stuff.c:1523	23%	4.01
quark_stuff.c:1460	5%	3.85
path_product.c:160	5%	3.1
path_product.c:128	18%	4.14
path_product.c:97	10%	4.15
path_product.c:73	7%	3.86
Dynamic Loops	Coverage	Speedup
quark_stuff.c:1420	7%	3.74
quark_stuff.c:1396	5%	3.95
quark_stuff.c:1375	14%	3.91
quark_stuff.c:1523	23%	4.01
quark_stuff.c:1460	5%	3.85
path_product.c:160	5%	3.1
path_product.c:128	18%	4.14
path_product.c:97	10%	4.15
path_product.c:73	7%	3.87

Table B.23: Loops parallelized for SJENG

Static Loops	Coverage	Speedup
moves.c:572	9%	1.67
search.c:199	12%	1.51
moves.c:460	10%	1.75
neval.c:432	5%	1.88
neval.c:493	24%	1.37
Dynamic Loops	Coverage	Speedup
moves.c:572	9%	1.63
search.c:199	15%	1.51
moves.c:460	10%	1.69
neval.c:432	5%	1.88
neval.c:493	21%	2.54

Table B.24: Loops parallelized for SPHINX3

Static Loops	Coverage	Speedup
approx_cont_mgau.c:279	48%	1.39
vector.c:513	35%	3.69
Dynamic Loops	Coverage	Speedup
approx_cont_mgau.c:279	48%	1.33
vector.c:513	35%	3.77