THE UNIVERSITY of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Complementing User-Level Coarse-Grain Parallelism with Implicit Speculative Parallelism

Nikolas Ioannou

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2012

Abstract

Multi-core and many-core systems are the norm in contemporary processor technology and are expected to remain so for the foreseeable future. Parallel programming is, thus, here to stay and programmers have to endorse it if they are to exploit such systems for their applications. Programs using parallel programming primitives like *PThreads* or *OpenMP* often exploit coarse-grain parallelism, because it offers a good trade-off between programming effort versus performance gain. Some parallel applications show limited or no scaling beyond a number of cores. Given the abundant number of cores expected in future many-cores, several cores would remain idle in such cases while execution performance stagnates. This thesis proposes using cores that do not contribute to performance improvement for running *implicit* fine-grain *speculative* threads. In particular, we present a many-core architecture and protocols that allow applications with coarse-grain explicit parallelism to further exploit implicit speculative parallelism within each thread. We show that complementing parallel programs with implicit speculative mechanisms offers significant performance improvements for a large and diverse set of parallel benchmarks. Implicit speculative parallelism frees the programmer from the additional effort to explicitly partition the work into finer and properly synchronized tasks. Our results show that, for a many-core comprising 128 cores supporting implicit speculative parallelism in clusters of 2 or 4 cores, performance improves on top of the highest scalability point by 44% on average for the 4-core cluster and by 31% on average for the 2-core cluster. We also show that this approach often leads to better performance and energy efficiency compared to existing alternatives such as Core Fusion and Turbo Boost. Moreover, we present a dynamic mechanism to choose the number of explicit and implicit threads, which performs within 6% of the static oracle selection of threads.

To improve energy efficiency processors allow for Dynamic Voltage and Frequency Scaling (DVFS), which enables changing their performance and power consumption on-the-fly. We evaluate the amenability of the proposed explicit plus implicit threads scheme to traditional power management techniques for multithreaded applications and identify room for improvement. We thus augment prior schemes and introduce a novel multithreaded power management scheme that accounts for implicit threads and aims to minimize the Energy Delay^2 product (ED^2). Our scheme comprises two components: a "local" component that tries to adapt to the different program phases on a per explicit thread basis, taking into account implicit thread behavior, and a "global" component that augments the local components with information regarding inter-thread synchronization. Experimental results show a reduction of ED^2 of 8% compared to having no power management, with an average reduction in power of 15% that comes at a minimal loss of performance of less than 3% on average.

Acknowledgements

First and foremost I would like to thank my supervisor Marcelo Cintra for his guidance and support during my PhD studies. He has been a tremendous adviser and a consistent source of insight and cool-mindedness. I can recall countless times when I would rush to him with a problem I had – either in conceiving a research topic or in implementing one – and he would always have the insight and background to swiftly understand the issue and provide invaluable feedback.

Second, I would like to thank my good friend Chronis Xekalakis who was a finishing PhD student when I began my studies and served as a mentor for me throughout my studies. He was patient enough to teach a novice how to do research in computer architecture and gave me the head-start into publishing papers by allowing me to participate in his own projects. I will always nostalgically look back to the countless inspiring discussions we had by the coffee machine at the Informatics Forum – sometimes quite loudly so – brainstorming about crazy research ideas or about our next paper. Outside the lab I had a blast hanging out with Chronis and the other guys, Georgios and Sofia, meeting new people, and having fun.

Additionally, I would like to thank Mike O'Boyle for running an inspiring and highly successful research group; it was a pleasure being part of. I would also like to thank Mikel Lujan for his help and collaboration, and his limitless hospitality during our visits to Manchester within the scope of our joint EPSRC grant. Moreover, I would like to thank Murray Cole for letting me be a teaching assistant for one of his courses for the last three years. He has always been patient and supportive.

I have been lucky to make several good friends during my PhD studies at Edinburgh. Salman was a senior student in our group when I started my studies and helped me, together with Chronis, in my first steps as a researcher and became an invaluable colleague afterwards. We spent several hours co-hacking the simulator (doing some "serious hacking" in our terminology) to implement the next big research idea. Fabricio and I started our PhD studies together under Marcelo and therefore had many common topics to discuss, when he decided to show up in the lab that is. Georgios (Tournabi), Sofia, Pedro, Lito, Vasilis, Karthik, Konstantina, George (Stefanakis), and Andrew have made my life easier and more fun during the time in Edinburgh and will miss very much them when I leave.

Everyone with a doctoral degree knows that the journey towards a PhD is full of ups and downs. Konstantina has always been there for me during the past four years, standing by my side and helping me overcome difficult situations. She has been a great source of stability, love, and support during both my undergraduate and graduate studies and can only hope that I have been able to provide her a fraction of that. Lastly, I owe huge gratidute to my elder sister, Anna, and my two wonderful parents, Yiannakis and Nina, who provided me with unconditional love and support for whatever I set out to do in my life.

Declaration

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

- "Complementing User-Level Coarse-Grain Parallelism with Implicit Speculative Parallelism"
  Nikolas Ioannou, and Marcelo Cintra
  International Conference on Microarchitecture, 2011

(*Nikolas Ioannou*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction, Contributions and Structure

With the shift toward multi- and many-cores, programmers can no longer enjoy steep performance improvements for free with every new generation of processors. Instead, parallel programming has to be employed both for programs written from scratch and for legacy code in order to exploit this new hardware. However, parallel programming is often hard and error prone, especially when addressing fine-grain threading which involves complex synchronization, communication, data partitioning, and scheduling [67, 96]. Thus, programmers often stay away from fine-grain parallelism and concentrate their efforts in exploiting parallelism at a coarser granularity. Coarse-grain parallelism [1] offers a good compromise between development effort and performance, and is often the first step exploited by programmers as they incrementally parallelize and performance tune their programs. Typical examples of such types of parallelism can be implemented via *PThreads* [84] and *OpenMP* [29].

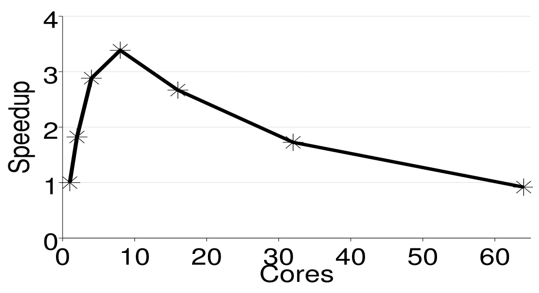
Given this focus on coarse-grain parallelism, applications often show limited or no scaling when executed in a large number of cores. Typical reasons for this are, among others, large critical sections leading to serialization in the presence of many threads, load imbalance between threads, and communication and coherence overheads [37, 46]. On the other hand, this focus on coarse-grain parallelism means that there is often room for opportunistically exploiting further degrees of fine-grain parallelism [37, 46].

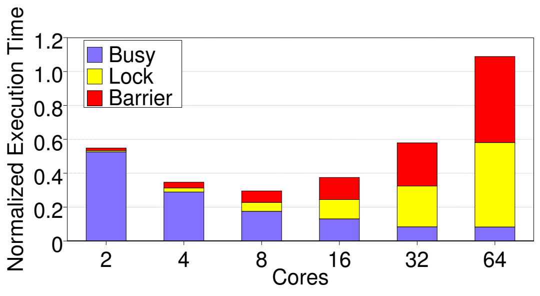
In this thesis we propose allocating cores beyond the application's scalability limit to exploit implicit speculative parallelism within individual explicit threads. Explicit

[1] We use the term coarse-grain parallelism to indicate parallelism under large tasks in terms of code size and execution time as opposed to fine-grain parallelism which means that individual tasks are relatively small in terms of code size and execution time.

```
/* Kernel */
Do for i=1 to Imax {
   if(master_thread)
      modify_the_sequence_of_keys;
   BARRIER(all_procs);

   for( i=0; i<NUM_KEYS; i++ )
      compute_the_rank_of_each_key_locally;

   lock(CS_lock);
   update_global_key_array;
   unlock(CS_lock);
   BARRIER(all_procs);

   if(master_thread)
      perform_partial_verification;
}
```

(a)





(b) (c)

Figure 1.1: Scaling behavior of the *is* benchmark from the NAS Parallel Benchmarks [7]: (a) Kernel source code, (b) Speedup scaling, (c) Breakdown of execution time.

threading is employed by conventional multiprocessors where the programmer explicitly specifies the partitioning of the program into threads and uses an application software interface and runtime system (like *PThreads* and *OpenMP*) to dispatch and execute multiple threads on several cores in parallel. Implicit threads, on the other hand, are transparent to the user and are either generated by the compiler or peeled off the sequential execution stream using hardware. By running implicit speculative threads through *thread-level speculation (TLS)* [45, 64, 78, 100, 104] performance can be improved beyond the application's scalability limit for a given input dataset. Moreover, given the guaranteed sequential semantics of the TLS protocol, this further parallelization is transparent to the programmers so that they do not have to struggle to further partition and debug the parallel code. In fact, with TLS it is possible to exploit whatever degree of parallelism exists within the coarse-grain explicit threads even in the presence of data dependences.

Figure 1.1 shows a simple example case study. Increasing the number of cores beyond eight causes performance to decrease significantly (Figure 1.1b). This is due to a large critical section whose relative execution time increases with the number of cores, as can be seen in Figure 1.1c. In the critical section (lines 10–12 in Figure 1.1a) each thread simply adds its local keys to the global key array in a *for* loop. This loop is amenable to parallelization and doing so could reduce the time spent in the critical section. Unfortunately, explicitly parallelizing this critical section requires writing some non-trivial code that allows threads to be dynamically detected when waiting at the critical section and then dynamically join the thread that is inside the critical section to assist it in performing its work.

Under our scheme the implicit speculative threads operate *within* explicit parallel threads with support for both types *simultaneously* in a nested fashion. Prior work that proposed architectures with support for both TLS and explicit threads [85, 86] could only accommodate either type at a time by switching between modes. We have developed a combined and nested coherence plus TLS protocol that provides coherence across explicit parallel threads and simultaneously provides TLS across multiple groups of implicit speculative threads, where each group is associated with a single explicit thread. This protocol is similar in spirit to previously proposed ones [81, 116] that allow nesting of transactional memory and coherence, but it requires some specific mechanisms in order to accommodate the differences in behavior between TLS and transactional memory.

Our approach has similar aims as Core Fusion [55] and Frequency Boost [50] in that they all attempt to pre-allocate or shift resources to a subset of cores in order to accelerate sections of a parallel code that do not scale well. However, they all differ in the source of the acceleration and hardware support. In our evaluation we quantitatively compare these approaches in terms of performance and energy efficiency.

In addition to improving the scalability of multithreaded workloads under implicit threads this thesis also focuses on the power management of such applications [2]. Workload imbalance between the threads of a multithreading application is a source of energy inefficiency. For example, under a fork-join parallel execution model like *OpenMP*, each thread performs its own portion of the parallel execution and then reaches a barrier at the joint point of the code that synchronizes all the threads. If all the threads reach the barrier the same time then synchronization stall time, and imbalance, is thus minimal. Often, however, threads reach the barrier at different times with some threads reaching earlier than others and spending significant time stalling. This discrepancy between thread arrival times is either due to heterogeneous tasks or due to execution performance variations like different cache behavior. Already substantial, we expect load imbalance to worsen as future CMPs expose more performance variations due to technology issues and thermal emergencies [15, 35]. Further, some multithreaded applications exhibit additional synchronization stalls induced by highly concurrent lock-based critical sections; this is another source of energy inefficiency.

In order to improve the energy efficiency of multithreaded applications due to synchronization stalls, prior work has used Dynamic Voltage and Frequency Scaling (DVFS) [74] to dynamically lower the frequency and voltage of cores running less critical threads with minimal performance impact [9, 19, 69, 71, 34, 53]. Most of these schemes focus on barrier-intensive applications [9, 19, 69, 71], however, and fail to report applicability to a larger subset of parallel workloads.

We, in turn, propose an adaptive, hierarchical power management scheme that aims at lowering the power consumption while maintaining the application performance, targeting a wide set of parallel workloads – barrier-intensive, lock-intensive and data-parallel ones. It comprises two components: (a) a "local" component that follows a thread's memory performance taking into account the difference in behavior between explicit and implicit threads and chooses a locally optimal voltage and frequency pair, and (b) a "global" component that tries to make globally optimal decisions based on the synchronization behavior.

1.1 Main Thesis Contributions

The main contributions of this thesis are:

- We are the first to evaluate implicit speculative parallelism *on top* of explicit

[2]We use the term scalability improvement throughout the thesis to indicate improvement in the performance of a multithreaded workload under a larger number of cores.

parallelism as a means to improve performance in traditional multithreaded applications that exhibit poor scaling.

- We discuss the architectural requirements for a system supporting implicit and explicit threads concurrently and evaluate such a many-core architecture.
- We present detailed analysis of performance bottlenecks in a set of multithreaded applications and evaluate their behavior in the presence of different input datasets.
- We present a hill-climbing approach that dynamically selects the number of explicit and implicit threads for a class of parallel programming style.
- We evaluate the amenability of accommodating implicit threads into traditional power management techniques for multithreaded applications. Furthermore, we expose some drawbacks of prior work on power management for parallel workloads.
- We present a new, hierarchical power management scheme for multithreaded workloads that both improves the state-of-the-art of power management and accounts for implicit threads.

Our experimental results show that complementing parallel programs with implicit speculative mechanisms offers significant performance improvements for a large and diverse set of parallel benchmarks. For a many-core comprising 128 cores, performance improves on top of the highest scalability point by as much as 102%, and 44% on average, for a system with 4 implicit threads per explicit thread and by as much as 85%, and 31% on average, for a system with 2 implicit threads per explicit threads. These performance improvements come with virtually no increase in total energy consumption. Compared to the alternative – Core Fusion and Frequency Boost – our approach often leads to higher performance with consistently lower energy consumption. Furthermore, our mechanism to choose the number of explicit and implicit threads performs within 6% of the static oracle thread selection.

Finally, our adaptive, hierarchical power management scheme significantly outperforms competing power management schemes on the evaluated platform and workloads and enjoys a significant reduction of Energy Delay^2 product of as much as 46% and 8% on average. This is due to a reduction in power consumption of as much as 47%, and 15% on average, with a minimal loss in performance of less than 3% on average. Significantly, our scheme maintains its applicability throughout all the types of parallel workloads evaluated – barrier-intensive, lock-intensive and data parallel alike.

1.2 Thesis Overview

This dissertation is organized as follows:

Chapter 2 provides background information. First, it presents the necessary background on TLS, in terms of its main concept, and architectural and compilation support. Second, it presents necessary background on Dynamic Voltage and Frequency Scaling (DVFS) on recent multi-cores as well as current trends in many-cores. Finally, it discusses prior work in power management for multithreading applications.

Chapter 3 discusses the general idea of exploiting fine-grain nested speculative parallelism to complement coarse-grain explicit parallelism to improve the scalability of multithreaded workloads. First, it presents the high-level semantics of nested speculative parallelization in multithreaded applications. It further presents the many-core architecture and protocols that implements these schemes. Moreover, it discusses a dynamic mechanism to automatically choose the number of explicit and implicit threads in *OpenMP* programs.

Chapter 4 describes the simulator setup and the benchmarks used for evaluation. It also discusses the system models evaluated, including core characteristics and on-chip network, as well as models evaluated for comparison purposes. Next, it discusses power management implementation details. Then, it presents the compilation framework, along with per workload information regarding speculation coverage and support.

Chapter 5 discusses experimental results. First, it provides an in-depth analysis of performance and scalability for the evaluated workloads. Second, it discusses the effect of different dataset sizes on scalability. Third, energy consumption results are presented and analyzed for the different schemes. Finally, the auto-tuning mechanism is evaluated and compared against static oracle results.

Chapter 6 presents our power management scheme *on-top* of the proposed implicit speculative parallelism scheme. First, it discusses room for improvement in prior work on power management for multithreaded workloads as a motivation to our work. Second, it presents the hierarchical, adaptive phase-driven scheme that augments the nested speculative parallelism to make it more energy efficient. Finally, it discusses the necessary hardware requirements to support our power management proposal.

Chapter 7 evaluates the proposed hierarchical power management scheme discussing experimental results. First, it discusses the amenability of the implicit threads to power management schemes. Second, it provides an analysis of the performance and energy consumption of our scheme against the current state-of-the-art power management schemes running explicit and implicit threads. Finally, it discusses the effectiveness of the proposed power management scheme under explicit threads only.

Chapter 8 discusses the related work and **Chapter 9** provides thesis conclusions and future work.

Chapter 2

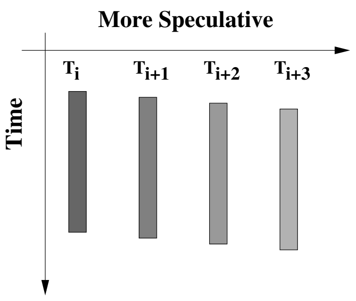
Background

This chapter provides the necessary background for the rest of the dissertation. The first part, in Section 2.1, discusses the high-level execution model for TLS as a means to auto-parallelize sequential applications. Section 2.1.2 presents the necessary architectural support together with compiler and task selection support for TLS. Section 2.1.3 presents the compiler support required by TLS. The second part, in Section 2.2, discusses the necessary background for power management. In Section 2.2.1 it provides background on Dynamic Voltage and Frequency Scaling (DVFS) on recent multi-cores as well as current trends in many-cores. Finally, in Section 2.2.2, it discusses prior work in power management for multithreading applications.

2.1 Thread-Level Speculation

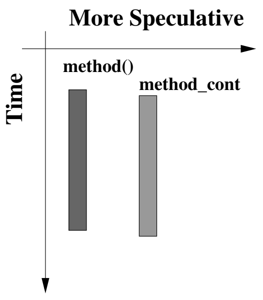
Under the *thread-level speculation* (also called *speculative parallelization* or *speculative multithreading*) approach, sequential sections of code are speculatively executed in parallel hoping not to violate any sequential semantics [45, 64, 78, 100, 104]. Sequential control flow imposes a total order on the threads. At any time during execution, the earliest thread in program order is *non-speculative* while the others are *speculative*. The terms *predecessor* and *successor* are used to relate threads in this total order. Stores in speculative threads generate unsafe *versions* of variables that are stored in a *speculative buffer*. If a speculative thread overflows its speculative buffer it must stall and wait to become non-speculative. Loads in speculative threads are provided with potentially incorrect versions. As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. Any value read from a predecessor thread is called an *exposed read*, and must be tracked since it may expose

```
for (i = 0; i < MAX; i++) {
  spawn_speculative_thread();

  /*loop body*/
  commit();
}
```



(a)

```
pid = spawn_speculative_thread();
if(pid != 0) {
  method();
} else {
  /*method continuation*/
}
```



(b)

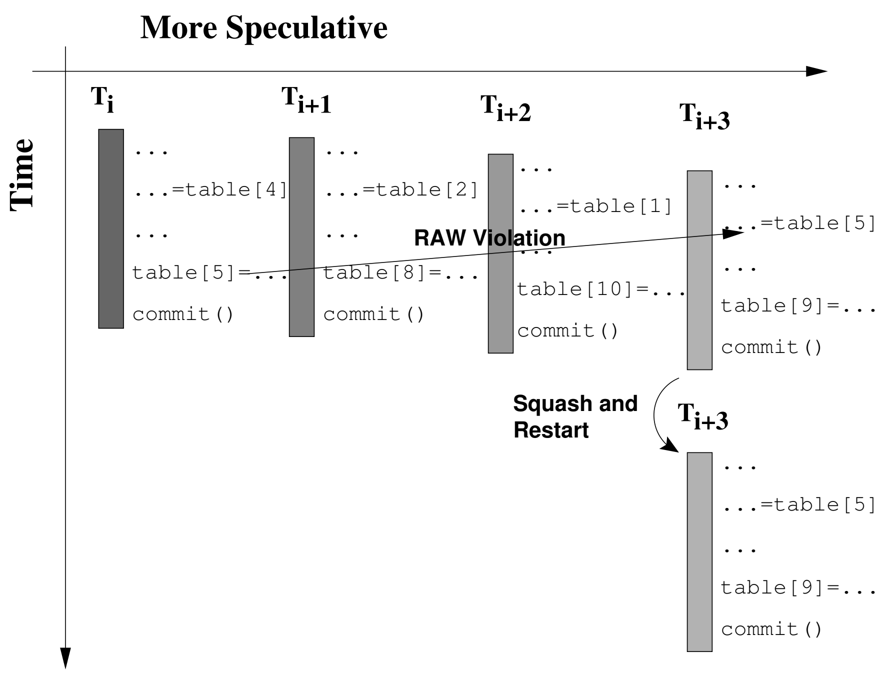
Figure 2.1: Speculative task extraction: (a) Loop-level speculation and (b) Method-level speculation.

a read-after-write (RAW) dependence. If a dependence violation is found, the offending thread must be *squashed*, along with its successors, thus reverting the state back to a safe position from which threads can be re-executed. When the execution of a non-speculative thread completes it *commits* and the values it generated can be moved to safe storage (usually main memory or some shared lower-level cache). At this point its immediate successor acquires non-speculative status and is allowed to commit. When a speculative thread completes it must wait for all predecessors to commit before it can commit. After committing, the core is free to start executing a new speculative thread.

Speculative threads are typically extracted from either loop iterations (Figure 2.1a) or method continuations (Figure 2.1b). The compiler marks these structures with a *spawn instruction* at the beginning, so that the execution of such an instruction leads to a new speculative thread, and a *commit instruction* at the end. The *parent* thread continues execution as normal, while the *child* thread is mapped to any available core. For loops, spawn points are placed at the beginning of the loop body, so that each iteration of the loop spawns the next iteration as a speculative thread. Threads formed from iterations of the same loop (and that, thus, have the same spawn point) are called *sibling* threads. For function calls, spawn points are placed just before the method call,

```
for(i=0; i<N; ++i) {
  ...
  a = table[index1];
  ...

  table[index2] = b;
  ...
}
```

(a)



(b)

Figure 2.2: Example of Thread-Level Speculation execution: (a) pseudo-code of a loop with infrequent dependences and (b) example of dynamic TLS execution.

so that the non-speculative parent thread proceeds to the body of the function call and a speculative child thread is created from the method's continuation.

2.1.1 Example

TLS allows the compiler to automatically parallelize portions of code in the presence of statically ambiguous data and control dependences, thus extracting thread-level parallelism between whatever dynamic dependences actually exist at runtime. To illustrate how TLS works, consider the simple *for* loop in Figure 2.2a which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the array *table*, assuming that the indexes cannot be statically computed. While it is possible that a given iteration will depend on data produced by a preceding iteration, these dependences may in fact be infrequent if the hashing function is effective. Hence a mechanism that could speculatively execute the loop iterations in parallel – while aborting and re-executing any iterations which do suffer dependence violations – could potentially speed up this loop significantly, as illustrated in Figure 2.2b. In this example, the program runs on a shared-memory multi-core, and some number of cores (four, in this case) have been allocated to the program by the operating system. Each of these cores is assigned a unit of work, or task, which in this case is a single loop iteration. When complete, each task attempts to commit its speculative work. In this case a read-after-write (RAW) data dependence violation is detected between task 1 and task 4; hence task 4 is squashed and restarted to produce the correct result, while tasks 1, 2, and 3 commit, thus successfully overlapping execution. This example demonstrates the basic principles of TLS.

2.1.2 Architectural Support

In order to support speculative execution that maintains sequential semantics, the hardware must provide at least the following functionality:

- A mechanism to dynamically detect true memory dependences between speculative tasks, in order to determine whether the sequential semantics have been violated.
- A mechanism for buffering speculative state so that it may be discarded when a violation occurs or safely committed in case of successful speculation.
- A mechanism to spawn new speculative tasks.
- A mechanism to commit data written by speculative tasks that did not cause any violations and merge them with main memory.
- A mechanism for squashing and re-executing speculative tasks that have caused a violation of the sequential semantics.
- A mechanism to maintain the speculative task ordering with respect to the sequential execution.

The following paragraphs discuss typical implementations of these mechanisms in prior work.

2.1.2.1 Detecting Data Dependences

To support thread-level speculation, we must perform the difficult task of detecting data dependence violations at run-time, which involves comparing load and store addresses that may have occurred out-of-order with respect to the sequential execution. These comparisons are relatively straightforward for instruction-level data speculation (within a single thread), since there are few load and store addresses to compare. For thread-level data speculation, however, the task is more complicated since there are many more addresses to compare, and since the relative interleaving of loads and stores from different threads is difficult to track. There are three possible ways to track data dependences at run time; for each option, a different entity is responsible for detecting dependence violations. First, a third-party entity could observe all memory operations and ensure that they are properly ordered–similar to the approach of the Wisconsin Multiscalar's address resolution buffer (ARB) [38, 100]. Such a centralized approach has the drawback of increasing load hit latency which would hinder the performance of non-speculative workloads. Second, the producer could detect dependence violations and notify the consumer. This approach requires the producer to be notified of all addresses consumed by logically-later tasks, and for the producer to save all of this information until it completes. On every store, the producer checks if a given address has been consumed by a logically-later task and if so, notifies that task of the dependence violation. This scheme has the drawback that the logically-earliest task must perform the detection, but we want the logically-earliest task to proceed unhindered.

A third approach is to detect data dependence violations at the consumer. In this approach, consumers track which locations have been speculatively consumed, and each producer reports the locations that it produces to the consumers. Hence a producer task that stores to a location must notify all consumer tasks that have previously loaded that location, so that the consumer tasks can verify that proper ordering has been preserved. The key insight is that this behavior is similar to that of an invalidation-based cache coherence scheme: whenever a cache line is modified that has recently been read by another core, an invalidation message is sent to the cache that has a copy of that line. To extend this behavior to detect data dependence violations, we simply need to track which locations have been speculatively loaded, and whenever a logically-earlier task modifies the same location (as indicated by an arriving invalidation message), we know that a violation has occurred.

2.1.2.2 Buffering Speculative State

Speculative tasks generate speculative writes which cannot be merged with the permanent state of the system unless the task commits. These writes are stored separately, typically either in the cache of the core running the task [45, 94, 104] or in a dedicated speculative store buffer [64, 100]. If the task successfully commits, the state is merged with system state (typically either with main memory or shared lower level caches). If it is squashed before it reaches completion, buffered state is discarded. A task only commits if it completes execution and becomes non-speculative. This ensures that tasks commit in order, thus preserving sequential semantics. Also, the hardware must provision for the case that the speculative buffer overflows; speculative tasks can either squash and re-execute or simply stall and wait until they are "safe" and can commit their speculative buffer. Garzaran et al. [41] provide a study of different approaches to speculative buffering along with their respective advantages and disadvantages.

2.1.2.3 Data Versioning

Each task has one version of each datum. If a number of speculative tasks are running on a system, each has a different version of shared data. On commit, versions are merged into system state in task order. Some proposals allow one version per core [44], while others support multiversioned caches and hence allow a speculative task to execute on a core even if commit is still pending for a previously executed task [42, 94]. To allow efficient execution in the presence of shared data, speculative systems also forward shared data from earlier threads to later threads.

2.1.2.4 Spawning Speculative Tasks

Spawning a new operating system thread is a process that in conventional architectures is typically fairly slow. In TLS systems, where speculative thread spawns are fairly frequent, this would impair performance. For this reason special support for fast spawning of threads is required. More specifically, in TLS systems when a thread encounters a thread spawn instruction, it creates a small packet containing the stack pointer, the program counter and some counters that have to do with the thread ordering. This packet is sent to an empty core which can start execution immediately after initializing its program counter and stack pointer accordingly. One option is to rely on register communication, as Multiscalar [100] does. A second option is to perform communication of live-ins through memory. The compiler ensures that all values that are live-ins for the newly created thread will be spilled into memory (through register spilling), so that when the new thread requests them they will be propagated to it via the TLS protocol.

2.1.2.5 Committing Speculative Tasks

We cannot determine whether speculation has succeeded for the current speculative task until all previous tasks (ordered based on the sequential semantics) have made their speculative modifications visible to memory – hence the act of committing speculative modifications to memory must be serialized. This could be done, for example, via an entity that maintains the ordering of the active tasks. A more scalable approach, however, is to directly pass an explicit token–which we call the safe token–from the logically-earliest task to its successor when it commits and makes all of its speculative modifications visible to memory. Receipt of the safe token indicates that the current task has no speculative predecessors, and hence is no longer speculative. This safe token mechanism is simply a form of producer/consumer synchronization and hence can be implemented using normal synchronization primitives.

Moreover, when a thread finishes execution, any cache lines it modified have to be written back to memory. This is typically done via a lazy policy, where lines that should be written back are left in the caches until they are replaced and thus written back to a lower level cache. It is worth noting that in the presence of multiversioned caches [42], execution of subsequent threads can proceed without having to wait for all the previous cache lines of the previous task(s) running on the same core to be committed to memory.

2.1.2.6 Squashing and Re-executing

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

2.1.3 Compiler Support

Thread Level Speculation requires some compiler support that typically involves task selection, code generation and TLS specific performance optimizations.

2.1.3.1 Task Selection

In most TLS systems proposed in the literature, task selection is done statically at compile time. Tasks are typically extracted out of high-level program structures, such as loop iterations and function call bodies. This static approach is used, for example, in the POSH [72], and Spice [90] compilers, as well as the compiler infrastructure used in the STAMPede [103] TLS system. Other systems, however, take a different approach and are not limited to tasks coming only from loops or function call bodies but encompass finer-grain tasks (e.g., at a basic block level). The Min-Cut approach to task decomposition [58], for example, applies graph theoretic algorithms to the control flow graph, such that all basic blocks and combinations of basic blocks are candidates for tasks. The Mitosis system [87] identifies spawning pairs and forms tasks out of them. These are pairs of instructions that meet certain conditions of control and data independence. More recently, the Anaphase system [75, 76] performs fine-grain task selection based on a graph partitioning technique which performs a decomposition of applications into speculative threads at instruction granularity leveraging communication and pre-computation slices to deal with inter-thread dependences. The tasks selected for the Multiscalar system [111] and the compiler framework by Bhowmik and Franklin [10] are also examples of fine-grain speculative task selection that do not rely on high-level structures alone.

2.1.3.2 Code Generation

Code generation can be done in two ways: (a) by adding the new TLS instructions as in-line assembly code using a source-to-source compiler (typically producing C code) and re-compiling the output through a traditional compiler or (b) by changing the back-end of a compiler infrastructure to output TLS code directly. For example, the STAMPede system [103] uses a source-to-source compiler on top of GCC, while the POSH [72] and the Mitosis [87] compilers directly produce TLS binaries.

2.1.3.3 TLS specific optimizations

Some compilers also perform TLS specific optimizations to make speculative execution more efficient. For instance, Zhai et al. [117] look at identifying dependent scalar use and define pairs and then aggressively scheduling the USEs late in the consumer task and the DEFs early in the producer task. Steffan et al. [102] have noted that small loop bodies can be made more TLS friendly through loop unrolling. Software value prediction code may also be inserted at compile time, as by Mitosis [87].

2.2 Power Management

2.2.1 DVFS

Dynamic power P_{dy} dissipated by a chip is strongly dependent on supply voltage V_{dd} and operating frequency f:

$$P_{dy} \propto V_{dd}^2 f \tag{2.1}$$

By reducing the voltage by a small amount, dynamic power is reduced by the square of that factor. However, reducing the voltage means that transistors need more time to switch on and off, which forces a reduction in the operating frequency. Dynamic Voltage and Frequency Scaling (DVFS) [74] exploits this relationship by reducing the voltage and the clock frequency when this can be done without experiencing a proportional reduction in performance. Reducing frequency can usually be done quickly, whereas for changing voltage the regulators have to settle their output voltage. Changes in voltage must, thus, be carefully scheduled in advance to align ramping up voltage with activity in the chip.

Adjusting the voltage and frequency is done by means of a DC-DC converter, which changes the voltage to the desired levels. The new operating voltage is then used to drive the frequency generator, which provides the chip with the operating frequency for the corresponding voltage level.

Most modern processors have support for DVFS in order to save power or to avoid thermal emergencies [51]. Experiments done in [32] show that it is advantageous to reduce the CPU frequency for a memory intensive task, but not for a CPU-intensive task. The performance of a task with high CPU utilization is linearly dependent on frequency, and thus will suffer significant throughput loss when the frequency is lowered. A memory intensive task, however, will suffer minimal performance loss when the frequency is reduced. If a task is constantly accessing memory, then the CPU is constantly stalling and waiting for memory. Power consumption can be reduced by lowering the frequency for a memory intensive task, and system performance can be increased by running a CPU-intensive task at the highest frequency.

2.2.2 Power Management for Multithreaded Workloads

Applications often do not show the same behavior throughout their execution, and they typically have a dynamic fluctuation in terms of instructions per cycle (IPC). These regions of repeatable behavior have been characterized as program *phases* [6, 33, 36, 99]. A low IPC typically means that the application is stalling on long latency events and is unable to effectively utilize the computing resources. This is usually due to memory-bound or bandwidth-bound phases but could also occur in phases with complex control flow that incur high branch misprediction rates. When stalling on long latency events, dynamically reducing the voltage/frequency of the core executing the application can result in a significant reduction in its leakage and dynamic energy consumption with a relatively small degradation in performance. Therefore, operating cores always at the maximum voltage/frequency is not always beneficial and a DVFS policy that dynamically adapts to the application's performance can achieve higher energy efficiency.

Prior work has applied DVFS to single cores [2, 16, 49, 77, 97], clusters of multi-chip multiprocessors [39, 70, 95, 101], shared-memory multi-cores [5, 9, 19, 56, 69, 71, 91], and many-cores [34, 53]. They focus on either sequential, multi-programmed or multithreaded workloads. We focus on multithreaded workloads running on a shared-memory many-core.

DVFS schemes for multithreaded applications have primarily targeted workloads using barriers [9, 19, 69, 71]. Barriers are widely used synchronization primitives in multithreaded applications. A barrier is essentially a mechanism to prevent the progress of threads beyond the barrier until all threads reach the barrier. A thread that reaches the barrier early must wait for all other threads to arrive at the barrier. Barriers are typically implemented as shared counters that are incremented in a critical region whenever a thread arrives at the barrier [80]. All threads waiting at a barrier continuously check the value of this counter and continue only when it becomes equal to the number of threads that must synchronize at the barrier. This requires threads to continuously read their local copies of the shared counter, doing no useful work and consuming leakage and dynamic energy. Energy overheads associated with barrier synchronization are proportional to the barrier wait times of threads. An imbalance in workload distribution across threads can result in large barrier wait times. One solution to this problem is to use barrier-aware DVFS. This ensures that, at each barrier, the voltages and frequencies of cores are dynamically scaled such that faster threads do not arrive at the barrier early, but instead, arrive at the barrier at around the same time as the slowest thread. Using this technique, the wait times of threads at the barrier are minimized, and as a result, redundant leakage and dynamic energy consumption spent at the barrier is avoided. In order to obtain energy savings using barrier-aware DVFS, the discrepancy in thread execution times between two barriers, and in turn the expected barrier stall times when all cores run at the maximum frequency, must be accurately predicted. Meeting points [19] follows this approach. It assumes strict Single Program Multiple Data semantics (e.g., traditional OpenMP) and places "meeting points" at the end of each parallel loop. Each thread monitors its current progress based on these meeting points, compares it against the other threads' progress and accordingly throttles down if it detects that it is further ahead in execution. Thread criticality predictors [9] dynamically monitor the cache misses of each thread and give higher priority to the thread that suffers the most misses. The insight behind this is that the thread that is most likely to arrive last at the barrier is the slower one, and the one suffering the most misses is an obvious candidate.

An alternative solution to this is to not perform dynamic voltage and frequency scaling but instead put to sleep threads that arrive early. The thrifty barrier [69] follows this approach and tries to predict barrier stall times and puts the threads that are predicted to stall longer than a threshold to sleep in order to minimize energy consumed at barriers.

Chapter 3

Exploiting Implicit Speculative Parallelism in Explicitly Parallel Applications

This chapter discusses the high-level idea of exploiting implicit speculative parallelism to improve the performance of multithreaded applications together with the motivation behind it and the internals of its implementation. First, in Section 3.1, the general idea is outlined together with a discussion of the main sources of scalability bottlenecks for the parallel workloads evaluated, and how implicit speculative threads could theoretically improve scalability by tackling these bottlenecks. Second, in Section 3.2, it proposes a viable tiled many-core architecture with support for nested implicit speculative threads. Third, in Section 3.3, the coherence and TLS protocol of the many-core architecture are discussed. Finally, in Section 3.4, it presents a dynamic scheme that automatically finds the scalability tipping point of a parallel application and then decides whether to employ implicit threads on top.

3.1 General Idea

The key realization that we exploit in this thesis is that explicitly parallel applications with user-level coarse-grain threads (explicitly declared threads in coarse-grain parallel programs) are often limited in scalability sooner or later. Further decomposing the threads into smaller tasks beyond that point is often very difficult or futile. As future many-cores are expected to have tens to a few hundred cores, many applications will have an execution scalability cap below the total number of cores available on chip.

```
#pragma omp parallel for
for (k = 0; k < KMAX; k++) {
  /* ... */
  begin_nested_speculation();
  for(l = 0; l < LMAX; l++) {
    spawn_nested_spec_thread();
    /* inner loop body */
    commit_nested_spec_thread();
  }
  end_nested_speculation();
  /* ... */
}
```

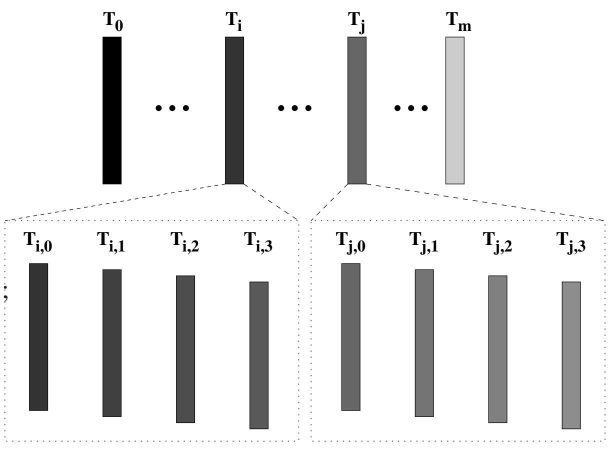


Figure 3.1: Implicit speculative parallelism on-top of an *OpenMP* parallel workload.

The key idea of this dissertation is to complement the explicit coarse-grain threads with fine-grain implicit speculative threads. In this scenario the programmer has to parallelize, debug, and performance tune the application only up to a desired cost-benefit point with coarse-grain threads. Further fine-grain parallelism is then exploited *from within each coarse-grain thread* by the hardware using thread-level speculation (TLS). As TLS provides sequential semantics, parallelism is exploited implicitly and transparently without the programmer having to worry about work partitioning, communication, synchronization, and scheduling. Finally, we note that since speculation is applied at the fine granularity of loops and procedure calls within a coarse-grain explicit thread, its effect is quite different from that of simply dividing the amount of work to be done by the explicit thread.

This concept of nested TLS within explicitly parallel threads is a natural expansion of traditional TLS, which was presented in Chapter 2. An illustration of nested speculation on-top of an *OpenMP* multithreaded loop is presented in Figure 3.1 and can be seen as an expansion to the example presented in Figure 2.1a. The nested implicit speculative threads within the explicitly parallel threads operate exactly as traditional TLS threads would within a sequential application.

3.1.1 Sources of Scalability Bottlenecks in Parallel Applications

Many parallel applications suffer from low scalability when executed on a large number of cores. In this section, we enumerate the primary reasons for this limited scalability on the workloads evaluated along with typical examples.

Large serial sections or critical sections that lead to serialization and thus prevent scalability is a first bottleneck. Amdahl's law [4] defines the importance of serial sections in parallel workloads and remains a first order constraint on coarse-grain parallelism, despite having been re-evaluated for massively parallel workloads [43] and for the multi-core era [46]. Even modern parallel workloads, like the *bodytrack* [1] benchmark from the PARSEC [11] benchmark suite, suffer from this textbook source of performance bottleneck limiting its scalability. The behavior of bodytrack is presented in Figure 3.2. The code snippet of Figure 3.2a shows the main kernel of the benchmark that includes serial elements that account for roughly 8% of its total sequential execution time. Amdahl's law allows for a theoretical 12.5 maximum speedup with an *infinite* number of cores. In practice this translates to even less performance due to parameters not taken into account by Amdahl (memory subsystem, interconnect, etc.), as shown in Figure 3.2b. The workload's scalability quickly stops at 16 cores and going beyond that yields insignificant benefits. While the main thread spends its time in serial sections, the remaining threads wait in spin-locks until the next parallel region is encountered. This is clearly illustrated in Figure 3.2c: for 2 cores the "Busy" time, which basically represents parallel region time, is cut *exactly* in half and the rest of the time is spent in "Lock".

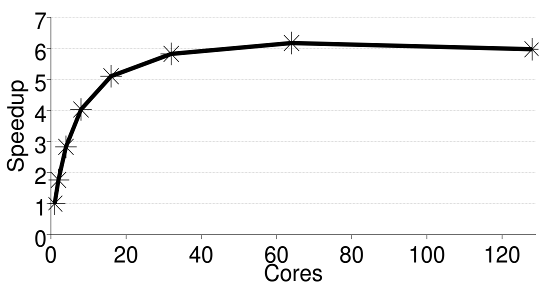
Second, load imbalance between the threads of a parallel application produces a source of scalability bottleneck for many applications. Parallel applications with load imbalance can be partitioned in two broad categories: (a) applications with task queues that involve heterogeneous tasks, and (b) applications with static large-grain work partition that leads to increased load imbalance. A common denominator of these parallel workloads is the increased time spent in barriers (or in task queue synchronizations), which typically increases exponentially as the thread count is increased.

The *radiosity* workload from the SPLASH2 [113] benchmark suite is an example of a load imbalanced parallel application with heterogeneous tasks (Figure 3.3). It is a task queue application with heterogeneous tasks that incur different loads in the workload's threads leading to flat scalability beyond 32 cores, depicted in Figure 3.3b. This is despite employing work stealing [12] to mitigate task imbalance, because its effectiveness is restricted by the limited length of the task window per iteration of the workload's main loop. Figure 3.3a shows the source code of the workload's main function that the threads execute in order to get a new task, illustrating the heterogeneity of

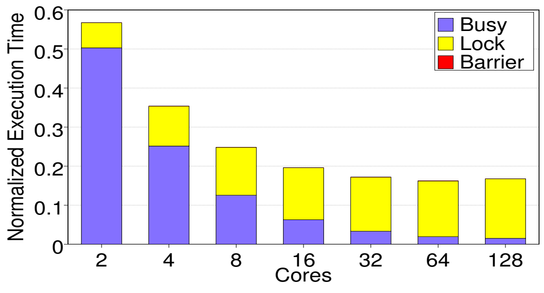
[1] We evaluate the *OpenMP* version of *bodytrack*, as discussed in Chapter 4

```
//loop over all annealing steps starting with highest
for(int k = (int)mModel->StdDevs().size() - 1; k >= 0 ; k--) {
  CalcCDF(mWeights, mCdf); //Monte Carlo re-sampling // *serial*
  Resample(mCdf, mBins, mSamples, mNParticles); // *serial*
  bool minValid = false;
  while(!minValid) {
    GenerateNewParticles(k);
    CalcWeights(mNewParticles);
    minValid = (int)mNewParticles.size() >= mMinParticles;
    //repeat if not enough valid particles
    if(!minValid)
      std::cout<<"Not␣enough␣valid␣particles-Resampling!!!"<<std::endl;
  }
  mParticles = mNewParticles;    //save new particle set
}
```

(a)



(b)



(c)

Figure 3.2: Scalability Bottleneck I: Large Critical Sections. Example using *bodytrack* workload from PARSEC [11]: (a) Source Code of benchmark's hot loop with the serial sections indicated by "*serial*", (b) Scalability graph, (c) Breakdown of Execution time.

the thread execution flow.

Figure 3.4 depicts the scalability behavior of the *swaptions* benchmark from the PARSEC suite. Despite being data parallel with respect to the number of "swaptions" [11] that are to be analyzed, it fails to exploit any other level of parallelism. If, for example, the number of "swaptions" is less than the number of available cores it simply does not yield *any* work for the remaining cores. This is depicted in Figure 3.4b with a dataset evaluating 16 "swaptions". One of the hot loops of this application is presented in Figure 3.4a and tries to compute random numbers required for the financial analysis later in the program. The number of iterations for each of the loop levels is *constant* (*BLOCKSIZE=16*, *iFactor=3*, and *iN=11*), independent of the thread count, and known at compile time. These iteration counts are typical for most of the workload's loops.

3.1.2 Use of Implicit Threads to Improve Scalability

Figure 3.5 illustrates the idea of using implicit speculative threads to attack each of the bottlenecks presented in the previous section to improve performance scalability. Figure 3.5a shows the case of an application dominated by critical sections. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 3.5a-i to Figure 3.5a-ii) successfully reduces the execution time of the parallel and critical sections, but overall scalability is limited by the serialization of the critical sections [2]. We propose to use the underutilized cores to run the critical section in TLS mode. By allocating resources for speculative threads (Section 3.2) we effectively trade off reduced execution time of the parallel sections for reduced serialization of the critical sections and possibly reduction of the execution time of the critical sections (Figure 3.5a-iii).

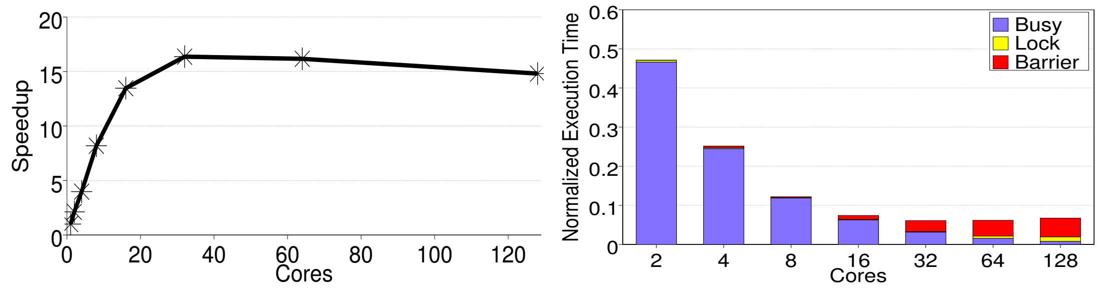
Figure 3.5b shows the case of an application where the parallel section is divided into one region with execution time proportional to the dataset and one with fixed execution time [3]. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 3.5b-i to Figure 3.5b-ii) successfully reduces the execution time of the dataset proportional region only. Again, by using the additional cores to run these sections in TLS mode we can achieve some parallelization of the non-scalable regions

[2]In some cases, as that of IS of Figure 1.1, the problem is worsened by the fact that the critical section time does not scale with the further work partition and the serialization problem is compounded with the fixed time of the critical section

[3]In reality the parallel threads might have several of these non-dataset-proportional regions spread throughout the thread. The example here is simplified for the sake of the explanation and our scheme is not limited in this way.

```
void process_tasks(unsigned process_id) {
  Task *t = DEQUEUE_TASK( taskqueue_id[process_id], QUEUES_VISITED, process_id );
 retry_entry:
  while( t ) {
    switch( t->task_type ) {
    case TASK_MODELING:
      process_model( t->task.model.model, t->task.model.type, process_id );
      break ;
    case TASK_BSP:
      define_patch( t->task.bsp.patch, t->task.bsp.parent, process_id );
      break ;
    case TASK_FF_REFINEMENT:
      ff_refine_elements( t->task.ref.e1, t->task.ref.e2, 0, process_id );
      break ;
    case TASK_RAY:
      process_rays( t->task.ray.e, process_id, process_id );
      break ;
    case TASK_VISIBILITY:
      visibility_task( t->task.vis.e, t->task.vis.inter,
      t->task.vis.n_inter, t->task.vis.k, process_id );
      break ;
    case TASK_RAD_AVERAGE:
      radiosity_averaging( t->task.rad.e, t->task.rad.mode, process_id );
      break ;
    default:
      fprintf( stderr, "Panic:process_tasks:Illegal_task_type\n" );
    }
    /* Free the task */
    free_task( t, process_id );
    /* Get next task */
    t = DEQUEUE_TASK( taskqueue_id[process_id], QUEUES_VISITED, process_id );
  }
  /* User Defined Barrier. While waiting for other cores to finish,
   * poll the task queues and resume processing if there is any task */
}
```

(a)

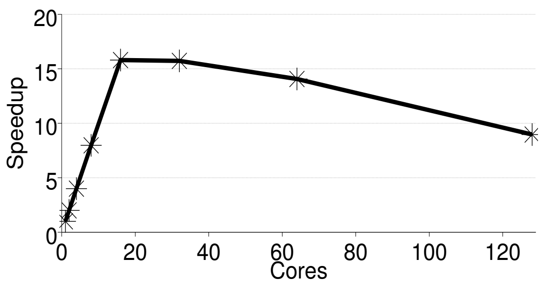


(b) (c)

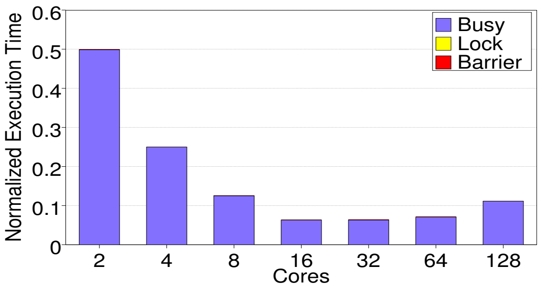
Figure 3.3: Scalability Bottleneck II: Load Imbalance. *radiosity* from SPLASH2: (a) Source code of the hot function *process_tasks()* with the FF_REFINEMENT and VISIBILITY tasks being modestly heavier than the others, (b) Scalability graph, (c) Breakdown of Execution time.

```
// =====================================================
// sequentially generating random numbers
for(int b=0; b<BLOCKSIZE; b++){
  for(int s=0; s<1; s++){
    for (j=1;j<=iN-1;++j){
      for (l=0;l<=iFactors -1;++l){
        //compute random number in exact same sequence
        randZ[l][BLOCKSIZE*j + b + s] = RanUnif(lRndSeed);
        /* 10% of the total execution time */
      }
    }
  }
}
```

(a)



(b)



(c)

Figure 3.4: Scalability Bottleneck III: Non proportional parallel sections. *swaptions* PARSEC [11] benchmark: (a) Source code of one hot loop, (b) Scalability graph, (c) Breakdown of execution time.

(Figure 3.5b-iii).

Another case we found in our experiments is that shown in Figure 3.5c of an application where static large-grain work partition leads to increased load imbalance. In this case, attempting to scale the application from 4 to 8 explicit threads (Figure 3.5c-i to Figure 3.5c-ii) successfully divides the work done but not in equal portions. By running the original explicit threads in fine-grain TLS mode we can achieve a more even partition of the work, leading to less load imbalance (Figure 3.5c-iii). Finally, we note that we can further improve on this simple model by exploiting implicit speculative threads also in the parallel sections of Figure 3.5a and in the dataset proportional regions of Figure 3.5b, as shown in Figures 3.5a-iv and 3.5b-iv respectively.

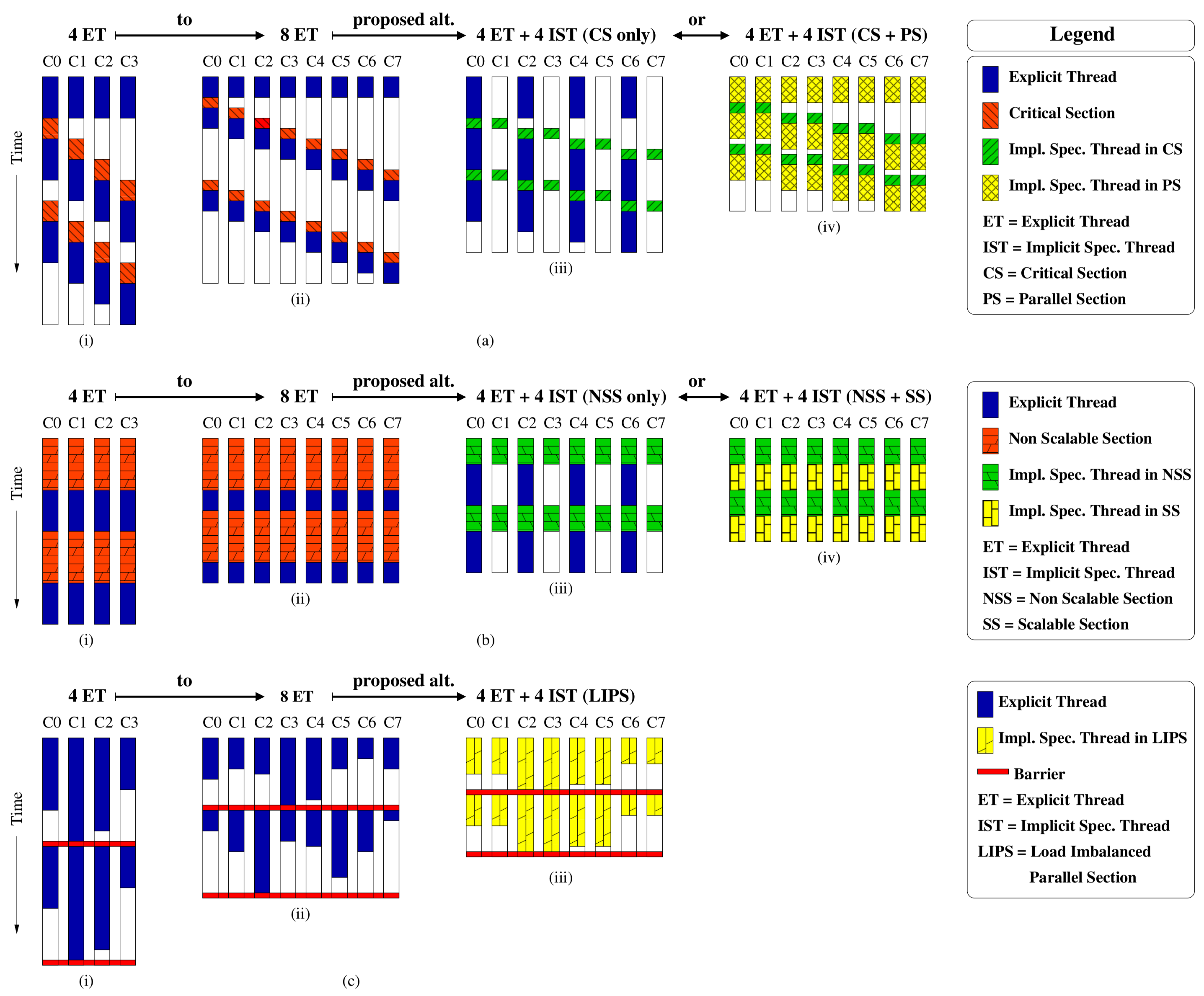


Figure 3.5: Sources of bottlenecks and possible uses of implicit threads: (a) Application with a large critical section, (b) Application with regions

3.1.3 Expected Performance Behavior

Figure 3.6 illustrates the performance behavior we hope to achieve with our proposed approach. In this example a baseline system with only explicit parallel threads scales in region **A** but stops scaling beyond 16 cores. In this region the speedups of both a 2-way and a 4-way TLS [4] schemes are likely below that of the baseline, since TLS, with its overheads and limited coverage, cannot compete with the performance gains from more explicit threads. However, after the baseline stops scaling, a 2-way TLS can potentially still provide performance gains for yet another doubling of the number of cores, as shown in region **B**. After this point, in region **C** we expect the speedup curve of the 2-way TLS system to behave similarly to that of the baseline, since a 2-way TLS can only provide at best a fixed performance boost over a corresponding baseline system with the same number of cores. On the other hand, a 4-way TLS performance curve will take longer to catch up with the baseline and a 2-way TLS system but, ideally, this system can still provide performance gains after the 2-way TLS stops scaling for yet another doubling of the number of cores, as shown in region **C**. Again, we expect that after this point (not shown in the figure) the speedup curve of the 4-way TLS system will also behave similarly to that of both the baseline and a 2-way TLS. Previous work on TLS has shown that it does not scale very well beyond 4 or 8 cores [54], which means that our proposed approach will only "buy" performance boosts up to systems with 4 to 8 times the number of cores. However, the goal of our approach is not to provide indefinite scalability, but to allow applications with poor scalability to better exploit the few hundreds of cores expected to become available in many-core systems by the end of the CMOS road map [14, 110].

Supporting this idea requires some small changes to the hardware. Unlike previous TLS schemes that attempted to achieve scalable performance solely through TLS, in our scheme it is sufficient that TLS be supported only in groups of small numbers of cores. In fact, if one expects most explicitly parallel applications to scale at least up to half or a quarter of the number of cores in the system, then all one needs is groups of 2- or 4-way TLS, respectively. In addition to supporting TLS within groups of cores, our scheme also requires some small changes to the TLS and coherence protocols in order to allow them to operate *simultaneously* in a nested fashion: coherence at the outer layer across groups of cores running explicit threads, and TLS at the inner layer within groups of cores running implicit speculative threads.

[4] We use the term n-way cluster (or n-way TLS) to refer to a scheme that partitions a given explicit thread into *n* implicit threads.

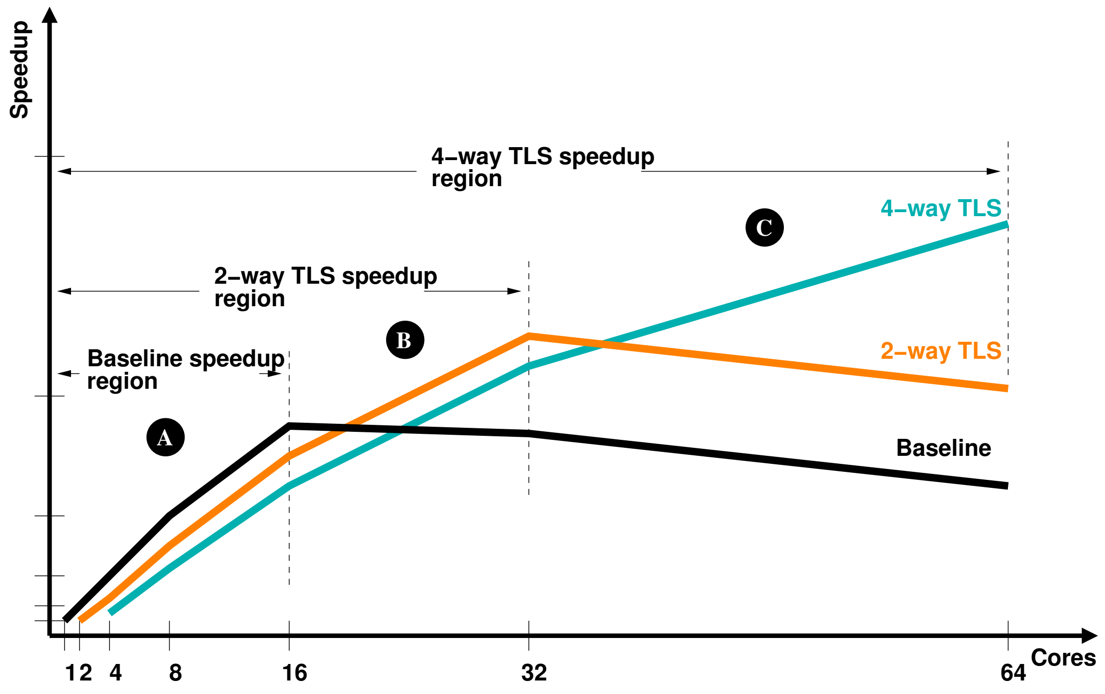


Figure 3.6: Expected speedup behavior with and without implicit speculative threads.

3.2 Many-Core Architecture

As explained in Section 3.1, our proposal to deal with the problem of limited scalability of explicitly parallel applications is to speculatively parallelize portions of the explicit threads. For this it suffices to support TLS within small groups of cores, or *TLS domains*. A natural physical organization is then to partition the many-core in several *tiles* as shown in Figure 3.7a, where each tile is an independent TLS domain, as shown in Figure 3.7b. Given the small number of cores per tile, cache coherence can be easily enforced within tiles by a snooping protocol on a bus, although directory based approaches are also possible. Cache coherence across tiles is enforced by a distributed directory protocol, which also interfaces with the intra-tile coherence protocol layer to build a fully coherent hierarchical system such as the one in [68]. This clustered organization is in line with expected trends for many-core systems [65] and can be already partially seen in the recent SCC system from Intel [47], which has multi-core tiles albeit without cache coherence and multi-level interconnects. The now canceled Rock processor [109] also features a clustered organization.

Each physical cluster corresponds to a TLS domain and TLS can be easily enforced with a snooping TLS protocol, such as [20], although directory-like approaches, such as [64], are also possible if coherence is also enforced by directories within clusters.

Again, there is no need to fully support TLS across clusters, although some interaction between the domains is necessary, as explained in more detail in Section 3.3.

3.3 Nested Coherence and TLS Protocol

As explained in Section 3.2, we base our architecture on a tiled many-core where each tile comprises a cluster of two or four cores, and where TLS is enforced within each cluster and coherence is enforced system wide. Simultaneously supporting coherence and multiple, independent TLS domains in a nested way imposes some restrictions on the flavor of TLS protocol used and also requires some mild changes to both protocols. The overall organization of the protocols is shown in Figure 3.7c, with a *TLS protocol layer* operating in each TLS domain underneath a *Coherence protocol layer* that operates across domains when running both explicit and implicit threads and additionally operates within domains when running only explicit threads.

3.3.1 TLS Protocol

The TLS protocol we use performs *eager conflict detection* at a mixed *word and line granularity* with *forwarding* and *lazy version management and commit*. An eager conflict detection policy (e.g., [94]) means that speculative threads are squashed as soon as a data dependence violation is detected. Conflicts occur when a speculative thread reads a value that is later modified. These can occur between a speculative thread and its predecessors in the same TLS domain (case ❹ in Figure 3.7c) and also between a speculative thread and the non-speculative thread in other TLS domains (cases ❺ and ❻ in Figure 3.7c). The later case has to be treated as a violation (leading to the squash of the speculative thread) to guarantee that a later speculative thread (in sequential order) does not consume an earlier value than an earlier speculative thread. Figure 3.8a depicts this problem: not squashing speculative thread $T_{i,2}$ at the time of the invalidation (action ❹ in the figure) leads to incorrect execution semantics. Let us walk through the steps of the example illustrated in Figure 3.8a to better understand this: first the safe thread $T_{i,0}$ in TLS domain TLS_i performs a load of memory location X and receives a value of 5 in action ❶. Second, the speculative thread $T_{i,2}$ from the same TLS domain performs a speculative load to the same location and receives the same value, namely 5 in action ❷. Third, the safe thread $T_{j,0}$ from TLS domain TLS_j performs a store to this location X in action ❸, and because this is a non-speculative thread this store is translated to a coherence invalidation message in action ❹. Finally, when speculative thread $T_{i,1}$, which is less speculative than $T_{i,2}$, loads the correct value of X (8) in action ❺. However, $T_{i,2}$ has already consumed a logically earlier value of X, therefore invalidating the sequential semantics. Thus, in the nested protocol incoming invalidation requests from other clusters lead to squashes of speculative threads (note that non-speculative threads, such as $T_{i,0}$ in the figure, do not have to be squashed). A similar approach to nesting is followed by transactional memory systems that enforce strong isolation between transactional and coherent threads [13].

We have chosen to implement conflict detection at the granularity of words within each TLS domain as this has been shown to provide better performance due to the absence of false violations leading to squashes [25, 86]. This requires that the partially updated cache lines merge in the order which the speculative threads commit. This in turn requires the ability to identify partial modifications. To this end, one bit per word within the speculative line indicates whether that specific word has been speculatively modified [103]. A speculatively modified cache line is committed by updating the current non-speculative state with only the words for which the modified word bits are set. However, for the conflict detection between speculative threads in one TLS domain and non-speculative threads in another TLS domain, as described above, we must perform conflict detection at the granularity of whole cache lines. This is because the coherence protocol operates at the granularity of lines. While the word level granularity of conflict detection has been shown to minimize the false violations in TLS [25, 86], the name cannot be said for cache coherence. The additional communication induced by word-level cache coherence and the hardware overheads needed at the directory structure and at the significantly larger last-level cache outweigh the benefits of reducing the false sharing, which can be largely achieved through compiler optimizations [108].

Forwarding [25] is supported entirely within a TLS domain when a speculative thread loads a value that has been previously modified by a predecessor speculative thread (case ❷ in Figure 3.7c). Speculative loads that do not find a version within the TLS domain must cross to the coherence layer (Section 3.3.2) in order to both obtain the return value and allow the identification of potential conflicts with coherent threads, as explained above (case ❶ in Figure 3.7c).

A lazy version management and commit policy (e.g., [20, 94]) means that writes by speculative threads are not merged with the non-speculative state until the thread becomes non-speculative and commits. In the nested protocol committing involves merging the state with the non-speculative state within a TLS domain and also with the

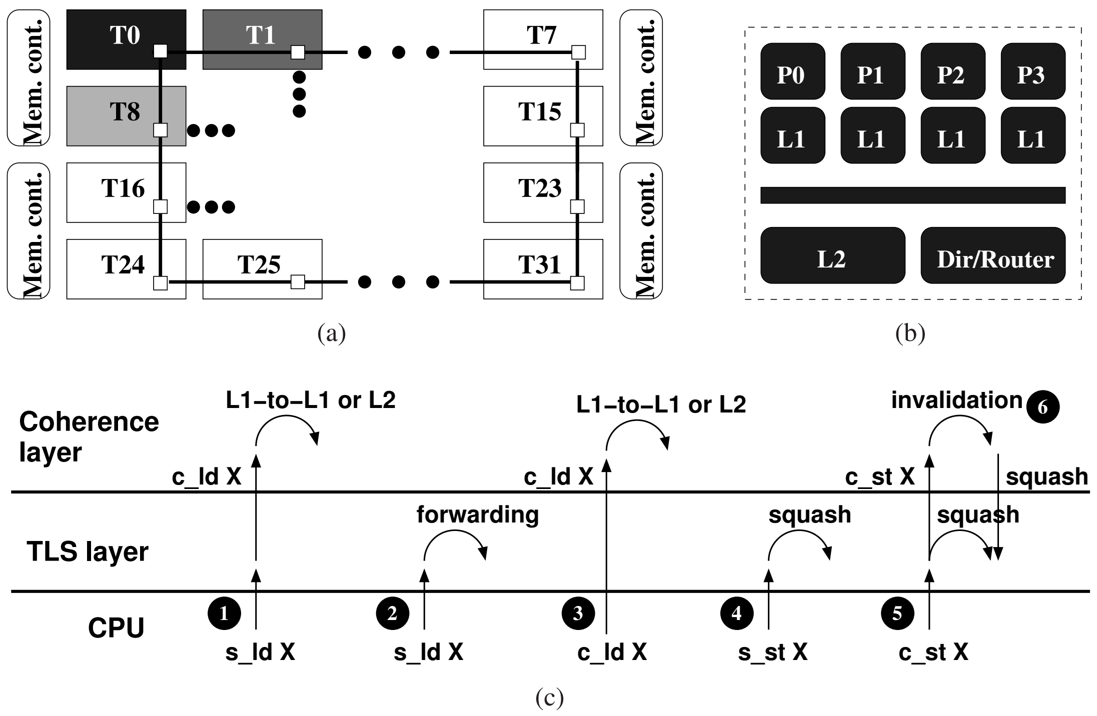


Figure 3.7: Organization of tiled many-core architecture: (a) Many-core organization, (b) Tile organization, (c) Hierarchical protocol view showing the following examples: A speculative thread issues a speculative load that gets translated to a coherence load in ❶, a speculative thread issues a speculative load that is forwarded through the speculative protocol bypassing coherence in ❷, a non-speculative thread issues a coherence load in ❸, a speculative thread issues a speculative store and a more speculative task in its TLS domain gets squashed in ❹, and a non-speculative thread issues a coherence store that results in a squash of one of the speculative tasks in its own TLS domain in ❺ but also to a squash of a speculative thread in another TLS domain through a coherence invalidation in ❻. Terms: s_ld: speculative load, s_st: speculative store, c_ld: coherence load, and c_st: coherence store.

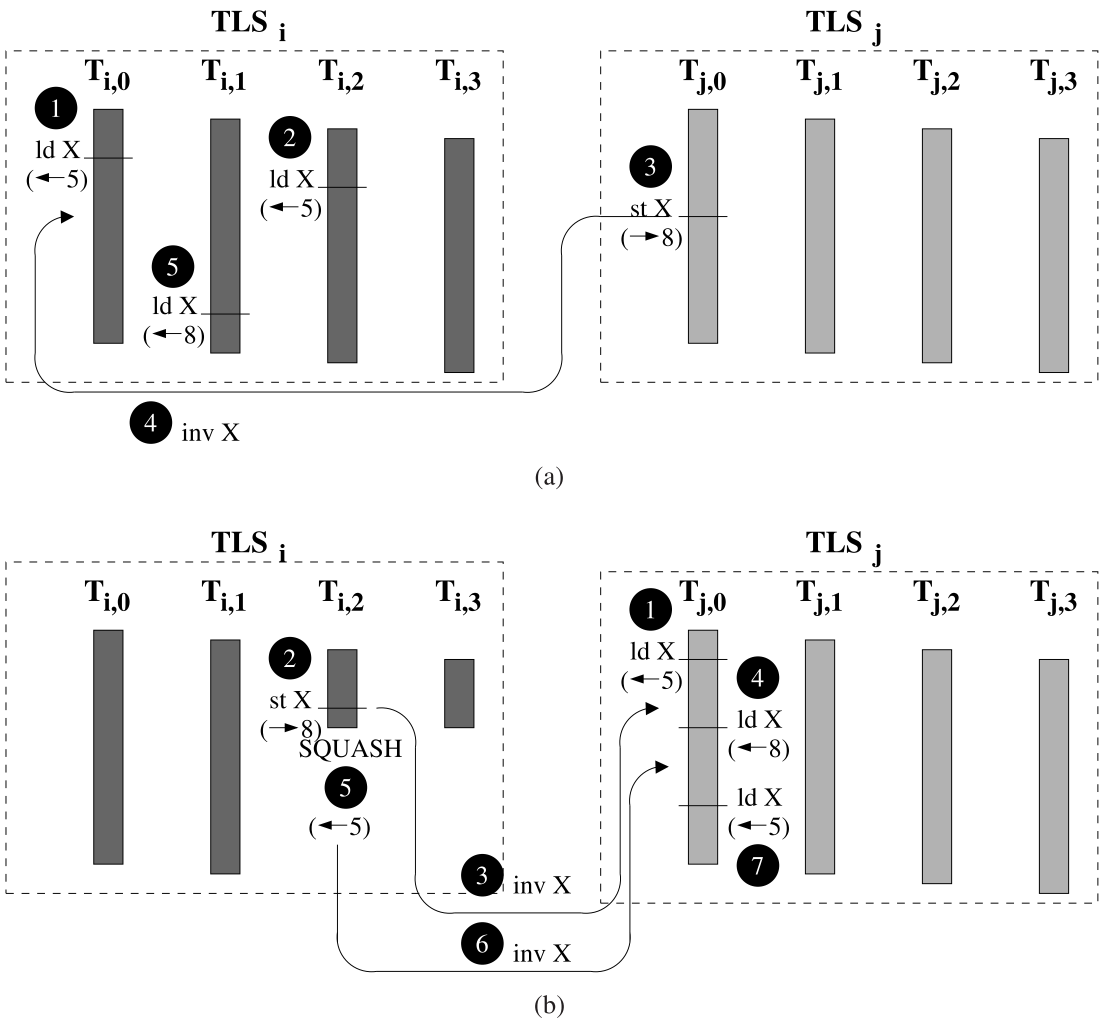


Figure 3.8: Example of interaction between coherence and TLS protocols: (a) Incorrect handling of coherence invalidation, (b) Incorrect log-based rollback.

coherent state across domains. We note that the alternative, eager version management with logs [82] leads to subtle interactions between speculative and non-speculative (i.e., coherent) threads as previously identified in [13]. The problem is depicted in Figure 3.8b: after a squash of thread $T_{i,2}$ undoing the write to X leads to an inconsistent state in the non-speculative thread $T_{j,0}$ since coherence transactions (invalidation ❸ following store ❷ in Figure 3.8b and subsequent load ❹ of the new value) cannot be undone. Let us analyze the steps in the example that lead to an inconsistent state. First, the safe thread $T_{j,0}$ in TLS domain TLS_j issues a load to memory location X and reads a value of 5 in action ❶. Subsequently, the speculative thread $T_{i,2}$ in TLS domain TLS_i issues a store to X, changing its value to 8 in action ❷, logging the previous value of 5 in case of rollback. Since we have eager version management this store, despite being speculative, translates to a coherence invalidation (action ❸). Thus, the subsequent load of X from thread $T_{j,0}$ in action ❹ will inevitably read this value of 8. Next, however, thread $T_{i,2}$ is squashed (action ❺), and the replay of its write log results in the invalidation in action ❻ for X restoring the value of 5. A further load of X from thread $T_{j,0}$ in action ❼ will yield a value of 5. It is thus clear that the intermediate value of 8 that was read by thread $T_{j,0}$ in action ❹ could lead to inconsistency. The lazy commit alternative avoids this by not allowing speculative stores to be communicated across clusters before thread commit. Finally, commits must appear to be atomic and, as mentioned above, involve not only the TLS domain but the entire system. Thus, lazy commit is reasonably straightforward in bus based systems (e.g., [20, 94]), but is more involved in directory based systems [21]. We follow an approach similar to the one in [21] where committing threads must obtain a special "commit token" from all directories associated with their read and write sets. After the token has been acquired the thread can safely commit all the cache lines in its write set, with the directories affected sending invalidation messages to any sharers. A speculative thread with the lower id always succeeds in committing when there is a write conflict with another speculative thread, thus avoiding deadlock.

3.3.2 Coherence Protocol

The coherence protocol used is *MESI* with *coarse-grain* directory state *per cluster* and a *"pseudo-CPU"* approach to interface *snooping coherence within each cluster* and *directory coherence across clusters* [68]. A coarse-grain directory with "pseudo-CPU" approach means that directories do not store coherence information for every core in the clusters, but only a summary information per cluster. This is convenient when running a coherent plus multiple speculative threads in each cluster since the TLS behavior does not have to be exposed to remote directories and the global coherence protocol. Instead, the “pseudo-CPU” (whose part is played here by the directory controller in each cluster - Figure 3.7b) is responsible for “translating” the remote coherence requests either into coherence transactions within each cluster when running only explicit threads, or into TLS transactions when running TLS within each cluster. Examples of these are cases ❶ and ❻ in Figure 3.7c: case ❶ is that of a load by an explicit thread or a non-speculative thread that is passed to the global coherence protocol, and case ❻ is that of an external incoming invalidation that is translated into squashes if a TLS violation is detected, as explained in Section 3.3.1. The local directories are also responsible for handling the speculative commit scheme of [21], as described in Section 3.3.1.

3.3.3 Speculative Buffering

Speculative state is buffered at the Level 1 caches [42, 25]. In the case of buffer overflow of a speculative thread, the thread simply waits until it receives the safe token. After it receives the token it can safely commit its speculative lines and resume execution. In an effort to reduce the TLS hardware overhead the speculative cache does not allow for multiple versions to co-reside in the same cache, unlike [42]. Only state from one speculative version, which is denoted by the core’s current speculative task id, is allowed to be alive at any point in time within the core’s Level 1 cache.

3.4 Dynamically Choosing the Number of Explicit and Implicit Threads

Finding the exact scalability tipping point for a particular application and input dataset of interest is commonly done in the HPC community by trials with increasing number of cores [31, 18]. Ideally, however, one would like a mechanism for automatically finding this tipping point on-the-fly and then choosing whether to employ implicit threads on top of explicit ones. For this purpose, we have developed a simple hill climbing algorithm (Algorithm 1) and implemented a prototype in the Omni OpenMP system [66]. This dynamic approach works for applications that are amenable to Dynamic Concurrency Throttling (DCT) [27, 28]. DCT, whereby the level of concurrency is adapted at runtime based on execution properties, is a software-controlled mechanism, or knob, for runtime power performance adaptation on systems with multiple cores. DCT cannot be applied to arbitrary parallel code regions without violating correctness. In principle, codes written in a shared-memory model where parallel computation does not include code dependent on the identifiers of threads, are amenable to DCT without correctness considerations. The vast majority of *OpenMP* codes meet this requirement for core-independence, as do the *OpenMP* workloads that we evaluate in this thesis.

The algorithm begins by choosing the initial number of threads to evaluate using a heuristic [5]: If the maximum number of iterations in all the *omp for* loops for the current parallel region can be determined statically and is less than or equal to the number of available cores (*MAX_CORES*, equal to 128 in the evaluated system) we set the initial number of threads to that value; otherwise we set it to 32 (number of available cores divided by 4) , which is a value that we empirically found to require the least amount of training time for the evaluated workloads (lines 2-6 in Algorithm 1). We then evaluate this initial thread count and perform a hill climbing search until it detects a slowdown in the execution time of the same parallel region or it encounters a thread count lower than 1 or greater than the number of available cores (lines 7-17 in Algorithm 1). Note that the hill climbing searches toward lower thread counts first as a repercussion of setting the initial thread count equal to the maximum iteration count. After settling on the number of explicit threads that yield the highest scalability, a simple empirical heuristic chooses whether to employ TLS or not (lines 18-19 in Algorithm 1). The heuristic is based on our observation that the tipping point on the number of explicit threads remains (mostly) unchanged when enabling TLS. Subsequently, the degree of TLS is chosen using the greedy heuristic presented in lines 18-19 of Algorithm 1, that gives priority to 4-way TLS if a sufficient number of cores is available.

[5] Note that we apply algorithm 1 to *each* parallel region, which usually consists of *for* loops that are executed several times.

Algorithm 1 Dynamically choosing the number of explicit and implicit threads via hill climbing

```
 1: for all omp parallel regions do
 2:   if $can\_determine\_max\_iter\_count\_statically \wedge (max\_iter\_count \leq MAX\_CORES)$ then
 3:     $orig\_cores \leftarrow cur\_cores \leftarrow 2^{\lfloor lg(max\_iter\_count) \rfloor}$
 4:   else
 5:     $orig\_cores \leftarrow cur\_cores \leftarrow MAX\_CORES \div 4$
 6:   end if
 7:   evaluate $cur\_cores$
 8:   repeat //start searching downwards
 9:     $evaluate\ cur\_cores \div 2$
10:   until $detect\_slowdown$
11:   $opt\_cores \leftarrow cur\_cores \cdot 2$
12:   if $opt\_cores = orig\_cores$ then // search upwards as well
13:     repeat
14:       $evaluate\ cur\_cores \cdot 2$
15:     until $detect\_slowdown$
16:     $opt\_cores \leftarrow cur\_cores \div 2$
17:   end if
18:   if $opt\_cores < MAX\_CORES \div 2$ then $enable\_TLS4$
19:   else if $opt\_cores = MAX\_CORES \div 2$ then $enable\_TLS2$ end if
20: end for
```

Chapter 4

Evaluation Methodology

This chapter discusses the experimental setup used throughout this thesis. First, in Section 4.1, it presents the choice of simulator and compiler. Second, in Section 4.2, it discusses the implementation of models and the choice of microarchitectural parameters. Third, in Section 4.3, it discusses the DVFS and power management implementation specifics. Finally, in Section 4.4, it presents the evaluated workloads and further discusses task selection, workload compilation, and profiling.

4.1 Simulation and Compilation Environment

We conduct our experiments using the SESC simulator [92]. SESC can model a variety of processor architectures, such as single processor systems, shared memory multicore systems, clusters of multi-chip multi-processors, and processors-in-memory. It models processor pipelines ranging from simple in-order ones to full out-of-order ones with branch prediction, multiple cache levels, network components, and several other components of modern processor systems. The MINT MIPS emulator is leveraged by SESC in order to emulate the MIPS instruction set architecture (ISA) and to generate instruction objects. These instruction objects are then used by the event driven SESC simulator for timing simulation. The access latencies of all the memory structures (e.g., register files, caches) were obtained using CACTI [107] for a 70nm technology. Note that the power and energy results reported later in the thesis and in Chapters 5 and 7 are always normalized to the baseline to which the respective schemes are compared against and not in absolute numbers. By showing relative improvements in terms of power and energy we in part alleviate the limitations of simulating a specific technology point and the results should extrapolate to future fabrication technologies, given that the dynamic power remains a significant part of the total power consumption. Power consumption numbers for the core and memory hierarchy are obtained using CACTI [107] and Wattch [17].

We generate the binaries using a version of GCC 3.4.4 specially modified to operate with SESC. The TLS binaries are generated through automatic instrumentation using the Cetus source-to-source compiler [30]. Selection of the speculative regions was done through manual profiling. We note, however, that this was done in the interest of time and does not constitute a major limitation as existing automated TLS profiling frameworks – such as POSH [72], which was not used due to its inability to handle explicitly multithreaded applications – have been shown to be very effective.

4.2 System Models

The main microarchitectural and system features are listed in Table 4.1. The system we simulate is a many-core with 128 cores, where each core is a 4-issue out-of-order superscalar akin to an Intel Core 2 [51]. The on-chip network is hierarchical, where cores within a cluster are connected via a snooping bus and clusters are connected via a point-to-point network. Contention is fully modeled at all levels of the interconnect and at the shared L2 caches. Figure 3.7 depicts the topology of the chip.

We have extended SESC to support our hierarchical hybrid snooping-directory MESI invalidation protocol (Section 3.3.2) and our variant of TLS (Section 3.3.1). All coherence and TLS transactions are handled in detail both in terms of functional and timing simulation.

For comparison purposes we also roughly evaluate two competing alternative systems: Core Fusion [55] and Frequency Boost, inspired by [50]. Core Fusion dynamically combines the computational resources of several cores together to deal with lowly-threaded workloads, while separating the cores in case of highly-threaded ones. Core Fusion is approximated by modeling a many-core comprising of wide 8-issue cores with all the core resources doubled (L1 caches, ROBs, Instruction Window, etc.) *and* without increasing the associated latencies. Thus, our model of Core Fusion is more aggressive than what can be implemented in practice and represents an *upper bound* of the performance that can be achieved with the technique. Frequency Boost is modeled as follows: for each idle core one other core gains a Frequency Boost of 800MHz (we assume a 0.2V increase in core voltage that results in the same power cap). Note that this is a static scheme where half the cores are switched off for the

Core	
Frequency	3GHz
Fetch/Issue/Retire Width	4/4/4
L1 ICache (IL1)	32KB, 2-way, 2 cycles
L1 DCache (DL1)	32KB, 4-way, 3 cycles
IL1/DL1 MSHR entries	10/16
IL1/DL1 block size	64B
I-Window/ROB	80/96
Branch Predictor	16Kbit Hybrid
BTB/RAS	1K entries, 2-way / 32 entries
Tile/System	
Number of Cores	128
Shared L2 Cache	8MB, 8-way, 13 cycles, 64B block size
L2 MSHR entries	64
Directory	Full-bit vector sharer list, 6 cycle latency
System bus transfer rate	48GB/s
Main Memory Access Latency	105ns
TLS	
Cycles to Spawn	20

Table 4.1: Architectural parameters.

Level	Frequency (GHz)	Voltage (V)
0	3.00	0.900
1	2.70	0.850
2	2.40	0.800
3	2.10	0.750
4	1.80	0.700

Table 4.2: Voltage-Frequency pairs.

entire execution of the application and the remaining half gain a constant boost in frequency. Intel's Turbo Boost [50] is a dynamic scheme which is enforced at large intervals when cores enter lower performance states. It is thus better applicable to multi-programmed workloads where cores are idle for long periods of time. In multi-threaded workloads like the ones used in this study, however, Turbo Boost would not be triggered since all cores are active all the time. Our static Frequency Boost policy is better suited for such scenarios. The shared portion of the memory subsystem, which includes the system bus, last-level shared cache, MSHRs, and on-chip interconnect, have the same parameters across the different configurations.

4.3 DVFS and Power Management

Additional requirements specific to power management are discussed in this section.

Each one of the tiles with its cores and their associated L1 caches form a separate voltage and frequency domain. This is in accordance with Intel's experimental Single-chip Cloud computer many-core [47] and can be extrapolated to future many-cores since the required circuitry for decoupling and converting voltages and frequencies on silicon may not scale as quickly as logic and memory transistors [63]. The shared L2 cache together with the interconnection network belong to a different domain (which is fixed at 3GHz at 0.900V). On-chip regulators are placed per tile so as to implement the different power domains, in a similar fashion to [63]. In order to synchronize communication between the distinct domains that operate asynchronously to each other we use the mixed-clock FIFO design proposed in [24].

We assume five voltage and frequency pairs, as shown in Table 4.2, similarly to the offered performance operation points in current commercial designs (e.g., the Super Low Frequency Mode, Low Frequency Mode, Normal Frequency Mode and High Frequency Mode used in [51]). All cores operate at the normal power mode (Level 0 in Table 4.2) except if our predictions dictate we should do otherwise. The cost for changing a power mode depends on the voltage swing and it is modeled to be 5 ns per 10mV in accordance with [63].

Furthermore, we model the three alternative power management schemes presented in Section 2.2.2 for comparison purposes. The Thrifty Barrier is modeled as described in the original proposal [69] with a slight modification: instead of going to sleep when predicting that the stall time will be big enough [1] we put the core to the lowest voltage-frequency setting. This has the disadvantage of lower power savings but minimizes the penalty of misprediction.

Meeting Points [19] is modeled as follows: since we already perform speculation in most of the workload's loops (see speculation coverage in Table 4.3) we treat the speculative commit instruction as "meeting points", in line with the original proposal. We compare each thread's current meeting point counter against the current maximum (i.e., the furthest meeting point reached so far) and decide whether or not to scale down, based on a thread's distance between this furthest meeting point. In order to mitigate frequent fluctuations in performance a 2-bit saturating counter per voltage frequency pair is maintained, in accordance with [19]. When a voltage frequency pair is suggested, its saturating counter is incremented by one and the rest of the pairs are decremented by one. If the suggested voltage frequency pair's saturating counter has saturated we then apply this new voltage frequency settings and do nothing otherwise. At the end of a barrier we simply reset each core's meeting point counters.

The original Thread Criticality Predictors scheme proposed in [9] assumes a 4-core multi-core with a monolithic last-level cache that has centralized visibility of the cache misses of all the cores. In the clustered many-core system that we evaluate, however, this assumption no longer holds. We thus assume a polling system instead, that periodically gathers each core's cache misses every 10K cycles (this is very aggressive in an effort not to hinder this scheme) and performs the power management algorithm presented in their original proposal [9]. We use the same 2-bit saturating counters, and criticality threshold values as presented in that paper.

[1]We have found that a prediction of 5K stall cycles amortizes the voltage change overhead of 300 cycles to change from the highest to the lowest voltage-frequency setting

4.4 Benchmarks

Benchmarks from the PARSEC [11], SPLASH2 [113] and the OpenMP C version of the NAS NPB (v2.3) [7] suites are evaluated. From PARSEC *blackscholes* (OpenMP version), *bodytrack* (OpenMP version), *canneal*, *streamcluster*, and *swaptions* are used. From SPLASH2 *cholesky*, *ocean-ncp*, *radiosity*, *volrend*, and *water-nsquared* are used. From the NPB, *ep*, *ft* [2], *is*, *lu*, and *sp* are used. Some of the benchmarks from the aforementioned suites were not included in our evaluation because they either: (a) either did not compile for our simulator infrastructure due to library incompatibilities or could not run with our infrastructure up to 128 threads due to reaching the simulator memory limits (*freqmine*, *dedup*, *facesim*, *ferret*, *x264*, and *vips*), (b) scaled all the way to 128 threads for a very small data set (*fluidanimate*), (c) showed similar scaling and bottlenecks with other benchmarks of the same suite (*barnes* and *radix* scaled similarly to *water*, *raytrace* similar to *radiosity*, *fmm* scaled close to *volrend*, *mg*, *cg*, *bt* scaled similar to *sp*), (d) were subsumed by benchmarks of other benchmark suites ($fft_{SPLASH2}$ and ft_{NAS}, $lu_{SPLASH2}$ and lu_{NAS}). For each of the benchmarks, we simulate the parallel region to completion. The parallel region is denoted by the *roi* segment for the PARSEC benchmarks, and by the main timer regions for the SPLASH2 and NAS benchmarks.

As mentioned earlier, TLS binaries were generated with the Cetus source-to-source compiler [30] and a modified version of GCC 3.4.4. The *OpenMP* benchmarks were first compiled using a modified version of the Omni OpenMP compiler and runtime system [66] before being provided as input to our compiler infrastructure. Speculative spawn and commit points were added to loops within hot functions. Compiler transformations to reduce data dependences, such as variable privatization, induction variable elimination, reduction variable expansion and min-max expansion are also performed automatically. Dependences in **rand** functions are relaxed to allow reordering of calls to the function due to the commutativity of such functions in *streamcluster*, *swaptions*, and *ep*. Furthermore, register spilling is done at task boundaries so that all inter-task register dependences are guaranteed to be communicated through memory. A single speculative run per benchmark for the sequential version and for a data set smaller than those evaluated was then performed to remove spawn points that slowed down execution. Detailed information about the benchmarks showing the speculation

[2] The results for *ft* are updated with respect to those published in [52]. The main NPB timer for *ft* included serial regions which we have excluded in these results to better reflect its parallel behavior.

types applied, the coverage of the speculative regions in terms of percentage of the program's sequential execution, and the datasets used is depicted in Table 4.3. Further information about the benchmarks can be found in Appendix A.

Benchmark	**Description**	**Input Sizes**		**Coverage of Speculative Regions**[a]	**Types of Speculation**
		Normal	Large		
PARSEC					
blackscholes	Financial Analysis	in_16K	in_64K	100%	LLS
bodytrack	Computer Vision	sequenceB_1	sequenceB_2	59%	LLS
canneal	Chip Design	100000.nets	200000.nets	99%	LLS
streamcluster	Data Mining	4K	8K	92%	LLS
swaptions	Financial Analysis	16	32	80%	LLS,MLS
SPLASH2					
cholesky	Sparse Matrix Multiplication	tk15	tk29	81%	LLS
ocean-ncp	Ocean Current Simulation	130	258	87%	LLS,MLS
radiosity	Graphics Rendering	test	room	69%	LLS,MLS
volrend	3D Volume Rendering	headScaledDown2	head	97%	LLS,MLS
water-nsquared	Molecular Dynamics	512	1000	99%	LLS
NAS OpenMP					
ep	Random Number Generator	1M	4M	100%	LLS,MLS
ft	3D FFT PDE	128K	512K	99%	LLS
is	Integer Sort	65K	1M	4%	LLS
lu	Matrix Multiplication	12^3	33^3	99%	LLS
sp	3D Fluid Dynamics	36	64	88%	LLS

[a] The coverage of speculative regions is reported for the sequential run.

Table 4.3: Simulated workloads. (LLS: Loop Level Speculation, MLS: Method Level Speculation).

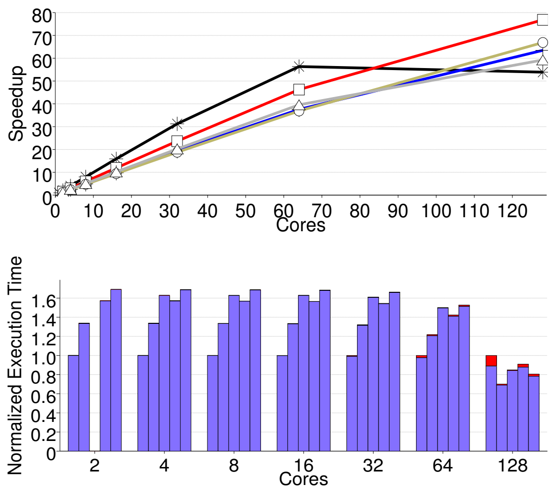
Chapter 5

Experimental Results

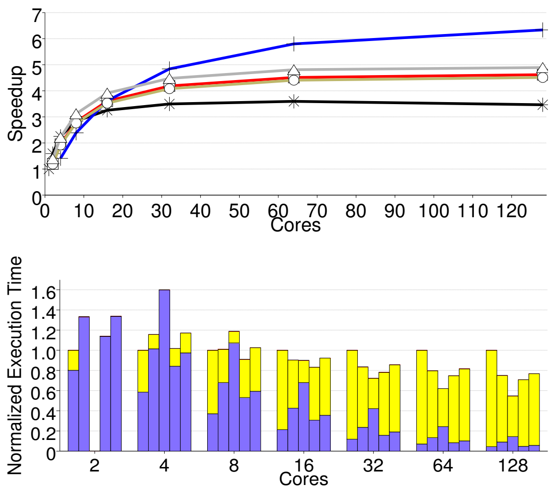
This chapter presents experimental results and performs a quantitative analysis on the performance and energy characteristics of the implicit speculative parallelism scheme presented in the previous chapters of the thesis. First, in Section 5.1, it provides an in-depth analysis of performance and scalability for the evaluated workloads. Second, in Section 5.2, it discusses the effect of different dataset sizes on scalability. Third, in Section 5.3, energy consumption results are presented and analyzed for the different schemes. Finally, in Section 5.4, the auto-tuning mechanism is evaluated and compared against static oracle results.

5.1 Performance and Scalability

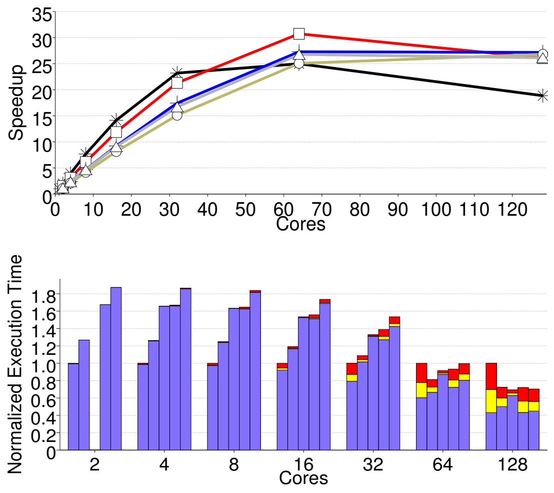
As explained in Section 3.1 our proposal is to employ implicit speculative threads to complement explicit user-level threads once performance with the latter stops scaling. The top plots for each benchmark in Figures 5.1, 5.2, and 5.3 show the performance of the different schemes as the number of cores is increased. The bottom plots show the breakdown of execution times according to busy versus synchronization (further divided into barrier and lock). The execution time bars are normalized to the baseline for the corresponding number of cores. For the baseline scheme the number of cores on the x-axes correspond to the number of explicit threads used. For Core Fusion and Frequency Boost the number of cores correspond to the equivalent amount of resources used, which are twice the number of explicit threads used (i.e., Core Fusion merges two cores to run a single explicit thread and Frequency Boost switches off one out of two cores to boost frequency of the active core). For our proposed scheme the number of cores on the x-axes correspond to the total number of explicit plus



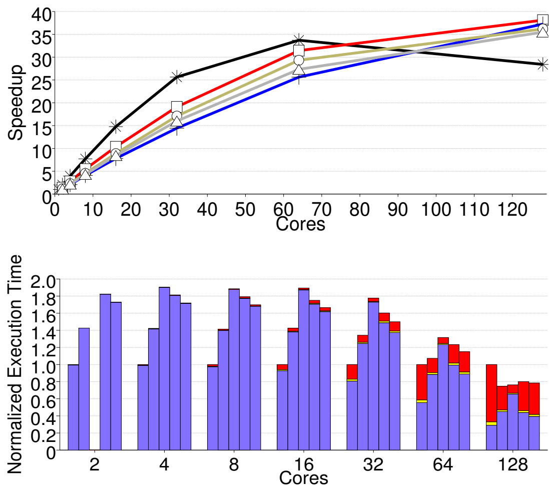
(a) blackscholes



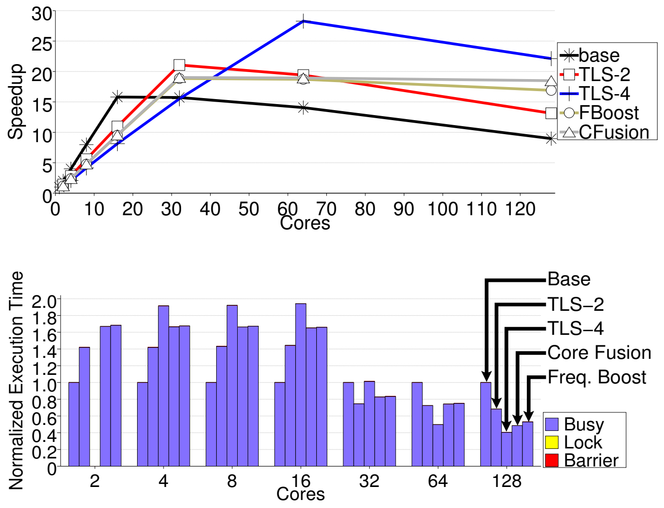
(b) bodytrack



(c) canneal



(d) streamcluster



(e) swaptions

Figure 5.1: Performance, scalability and bottleneck breakdown. (part I: PARSEC)

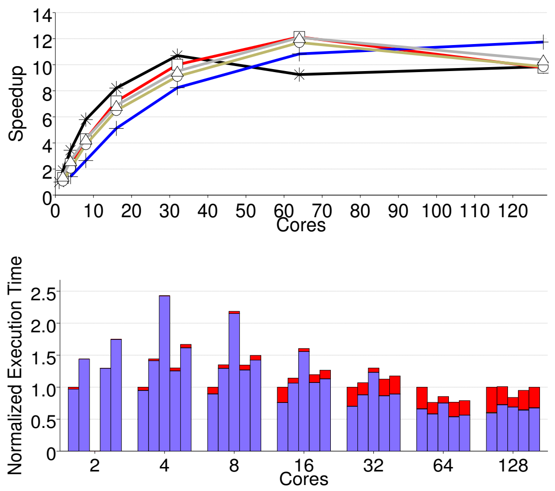
implicit threads used. Finally, for the baseline, Core Fusion, and Frequency Boost schemes, in the experiments with fewer threads than the number of cores in the system (which is fixed at 128) we report results for the best mapping of threads to tiles. For our proposed scheme we always map one explicit thread and its associated implicit speculative threads to each tile.

In the following we discuss each benchmark separately.

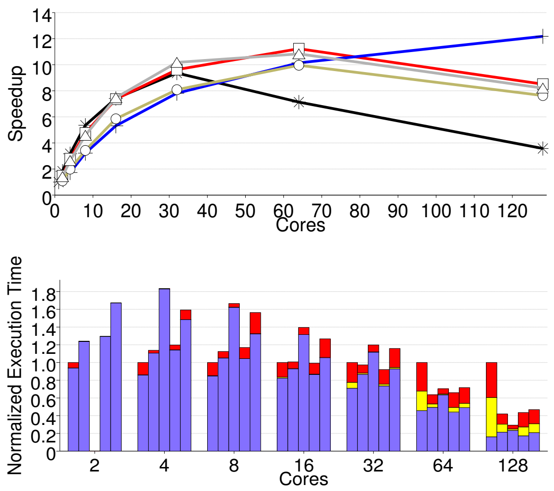
5.1.1 Detailed Analysis

blackscholes (Figure 5.1a) is a data parallel application that scales linearly up to 64 cores, but shows a small slowdown going from 64 to 128 cores. This is due to increased contention of the shared resources (last level cache and off-chip bandwidth) as well as increased barrier time. The TLS version of the benchmark operates on the main loop of *bs_thread()*, which accounts for all the program's parallel execution, by further partitioning its iterations to speculative tasks. The 2-way TLS improves the scalability by spending less time in barriers and by performing some prefetching to the shared cache. The 4-way TLS, however, offers only minor performance benefits; despite the similarly reduced barrier time and prefetching effects, it suffers from increased Level 1 data cache misses [40]. Frequency Boost is able to improve scalability due to a relatively high Instructions Per Cycle (IPC) of around 1.07 on average for the baseline version with 64 cores. Core Fusion is bounded by the relatively high branch misprediction rates that do not allow for performance improvements when going from 4-way to 8-way out-of-order superscalar.

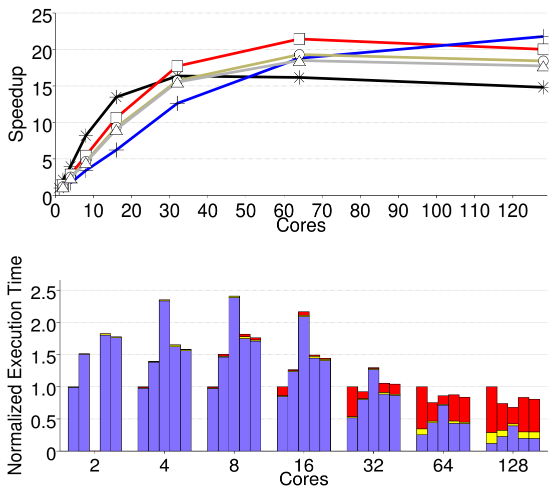
bodytrack (Figure 5.1b) exhibits poor scaling due to serial sections of code between parallel regions [11]. The increasing importance of those serial sections and the imposed load imbalance is reflected in the increased time spent in locks. By speculatively parallelizing loops in the serial sections as well as loops in the parallel sections we are able to improve its scalability. Speculated loops include loops in methods *ImageMeasurements::EdgeError()*, *ImageMeasurements::InsideError()*, *Projecte-Cylinder::ImageProjection()*, *FlexFilterRowV()*, and *FlexFilterColumnV()*. The high iteration count of the loops that we speculate upon reflects in the large 4-way TLS performance improvement over 2-way. Core Fusion and Frequency Boost also report significant speedups over baseline. This is due to faster execution of the serial sections. Core Fusion is more successful due its ability to better exploit the relatively high ILP found in some of the hot loops of the benchmark.



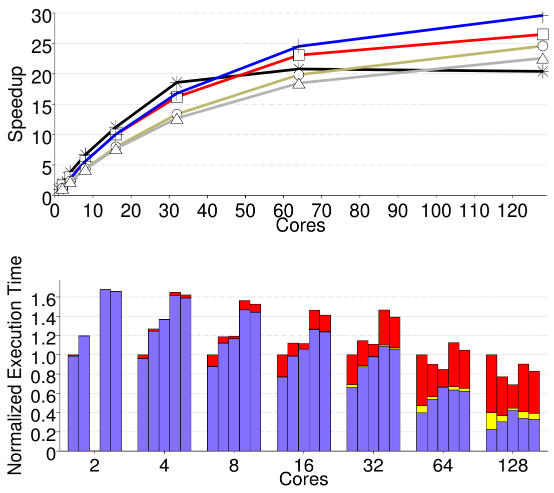
(a) cholesky



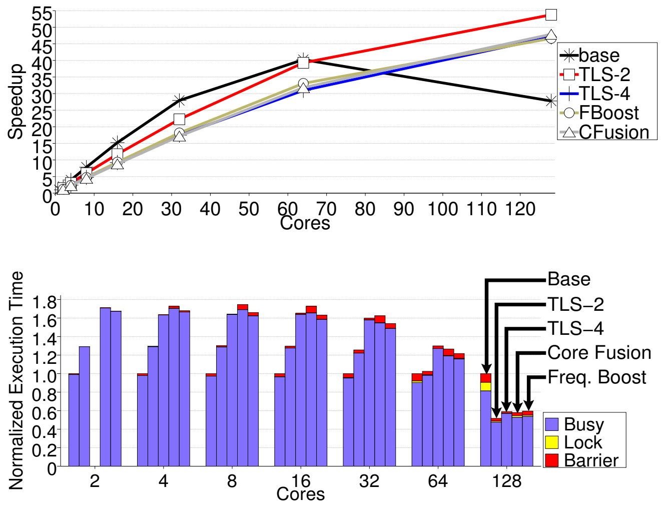
(b) ocean-ncp



(c) radiosity

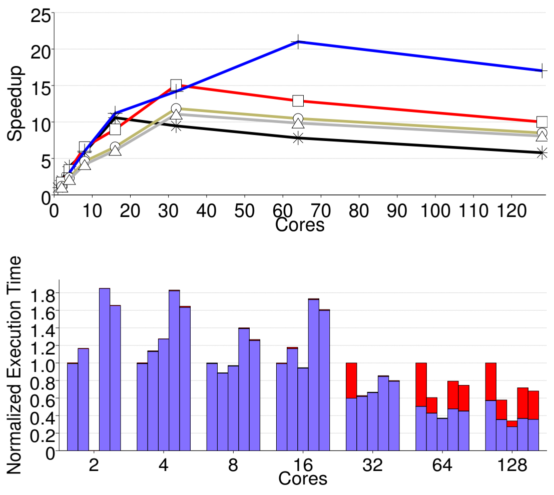


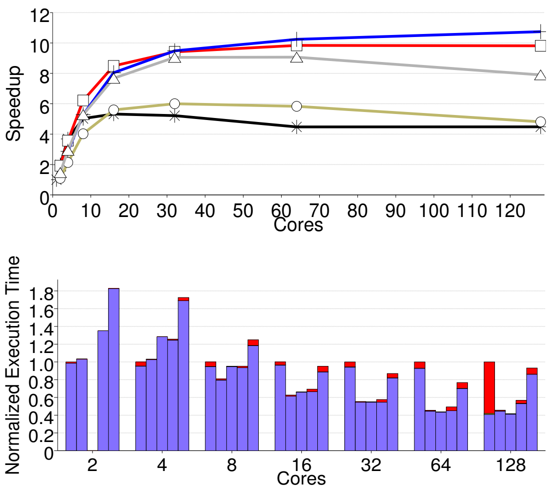
(d) volrend

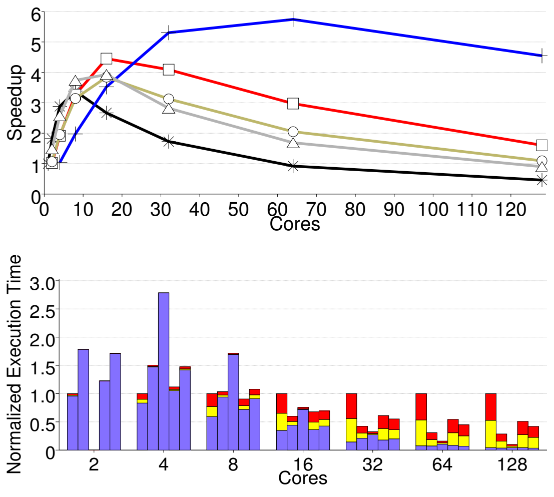


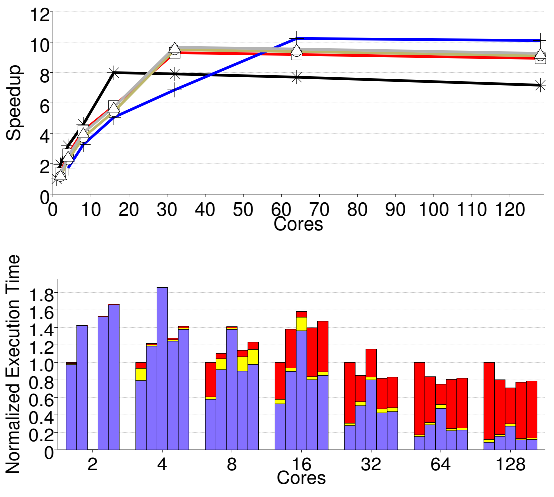
(e) water-nsquared

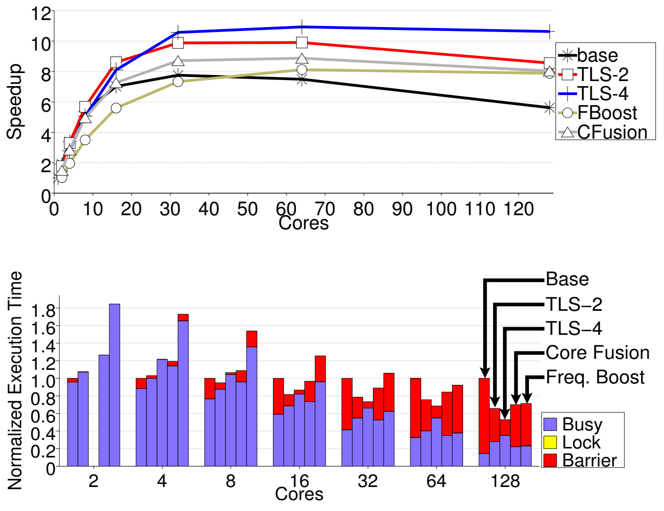
Figure 5.2: Performance, scalability and bottleneck breakdown. (part II: SPLASH2)


(a) ep


(b) ft


(c) is


(d) lu



(e) sp

Figure 5.3: Performance, scalability and bottleneck breakdown. (part III: NASPB)

canneal (Figure 5.1c) exhibits almost linear scaling up to 32 cores; going beyond that yields no further performance improvement. This is due to increased contention to the lock in the *Rng* object constructor and increased barrier time as we increase the number of cores, and due to high cache miss ratio and relatively high inter-thread communication [8]. We speculatively parallelize the hot loop in the *annealer_thread:Run()* method which takes most of the program's parallel execution time (Table 4.3) and are able to improve scaling to 64 cores. Despite the high-trip count of the hot loop and its high coverage, the TLS speedup, and especially the 4-way one, is only modest as the application becomes memory bound at this point. Moreover, the mediocre 4-way TLS performance versus 2-way is further attributed to the introduction of squashes in the 4-way case (Figure 5.4). Core Fusion and Frequency Boost offer minor performance improvements, again due to the memory-boundedness.

streamcluster (Figure 5.1d) exhibits good scaling up to 64 cores, but shows a slowdown when going from 64 to 128 cores. As the number of cores is increased the amount of work done by each thread decreases, except for some constant work done by the master thread at the beginning of the main kernel and in some of the steps of the *pgain()* function. This incurs load imbalance and manifests as increased barrier time. We speculatively parallelize most of the loops in the *pgain()* function, as well as the calls to *random()*, thus reducing load imbalance and improving scaling. The significant mispeculation rates for this benchmark, and especially for 4-way TLS, which are depicted in Figure 5.4 explain the lukewarm scalability boost. Both Core Fusion and Frequency Boost perform comparatively to TLS.

swaptions (Figure 5.1e) scales well with respect to number of *swaptions* in the input [11]. If the number of *swaptions* to be priced is less than the number of available cores the remaining cores are not utilized. The amount of computation done for each input *swaption* is constant and the loops are amenable to speculative parallelization and account for a significant portion of the program's execution time. Moreover, the speculation coverage is further increased by exploiting the permutability of the function call *RanUnif()*, which accounts for 10% of the total execution time. This translates to substantial performance increase for the 2-way and 4-way TLS versions. Both Core Fusion and Frequency Boost are able to attain increased performance through higher ILP, albeit significantly less than the TLS versions.

cholesky (Figure 5.2a) scales poorly and only up to 32 cores. This can be attributed to the high fraction of time spent on synchronization points, a relatively high cache miss rate, and high communication-to-computation ratio [113]. By speculatively parallelizing the loops in functions *ModifyTwoBySupernodeB()*, *ModifyBySupernodeB()*, *OneMatMat()*, *OneDiv()*, and *OneLower()* we are able to improve the application's highest scalability point, albeit by a small margin. This is primarily due to a relatively high mispeculation rate, especially for 4-way TLS (Figure 5.4), and secondarily due to being relatively memory bound. Core Fusion and Frequency Boost also offer marginal improvements to scalability.

ocean-ncp (Figure 5.2b) is characterized by intense usage of barriers and fine grain locks, as well as a relatively high cache miss ratio [113]. This limits its scalability and in fact causes a slowdown when going from 32 to 64 cores. We speculatively parallelize most of the loops in the *slave2()* function, effectively overlapping calls to *laplacalc()* and *jacobcalc()*. Loops in the functions *relax()*, *rescal()*, and *intadd()* are also speculatively parallelized. This speeds up each of the computational steps and thus reduces the time spent in the barriers that succeed each step. Moreover, the well known prefetching side-effect of TLS [72, 115] further improves performance by issuing misses early. Most loops speculated upon have a constant trip count of 2 which is reflected in the lack of performance improvement of the 4-way TLS version over 2-way. Core Fusion's improved tolerance of long latency events enables significant performance improvements close to that of 2-way TLS. Frequency Boost, on the other hand, increases the last-level cache miss latency in terms of core cycles and thus performs unfavorably to the other schemes.

radiosity (Figure 5.2c) uses tasks managed by distributed task queues. These tasks vary in size and, despite the use of task stealing, entail load imbalance beyond 16 cores, thus limiting the application's scalability. This is evident from the increased time spent in the user defined synchronization (barrier) as the number of cores is increased. By speculatively parallelizing the hot loop in the *compute_visibility_values()* function in the highly significant *visibility_task* and the calls to the *compute_form_factor()* function in the also hot *ff_refinement_task* we effectively reduce load imbalance, thus allowing the application to scale better. 4-way TLS offers little benefits over 2-way in this benchmark due to the introduction of squashes when going from 2-way to 4-way TLS, which is depicted in Figure 5.4. Both Core Fusion and Frequency Boost are able to attain increased performance through higher ILP, albeit significantly less than the TLS versions.

volrend (Figure 5.2d) is a barrier intensive workload due to serial sections at the beginning of the rendering process (*Render()*) as well as at the epilogue of its main loop (*WriteGrayScaleTIFF()*). These serial sections account for 3% of the sequential run and quickly become the bottleneck beyond 32 cores. The rest (97%) of the parallel region is spent mostly in the main loop of *Ray_Trace_Non_Adaptively()* entailing calls to *Trace_Ray()* as well as loops in other functions (*Pre_Shade(), Multiply_Matrices, etc*). By speculatively parallelizing the calls to *Observer_Transform_Light_Vector()* and *Compute_Observer_Transformed_Highlight_Vector()* at the serial preamble of *Render()* TLS is able to partially reduce the time spent in serial sections and improve scalability [1]. The rest of speculatively parallelized loops have a high-trip count that translates to improved TLS-4 scalability. Frequency Boost and Core Fusion provide little performance benefits because they do not reduce the barrier time, which is the primary factor that limits the scalability of this workload.

water-nsquared (Figure 5.2e) scales very well up to 64 cores, but beyond this point there is a slowdown due to additional synchronization and communication. The hot loops in functions *INTERF()* and *POTENG()* that perform most of the computations for each molecule show infrequent data dependences and are amenable to speculative parallelization. We additionally parallelize speculatively loops in functions *INTRAF()*, *PREDIC()*, and *CORREC()* thus covering most of the application's parallel execution (Table 4.3). This translates to a significant speedup of the 2-way TLS over the best performing baseline, scaling all the way to 128 cores. The 4-way TLS performs unfavorably compared to the 2-way TLS due to a large increase in the number of squashed threads (Figure 5.4). Core Fusion and Frequency Boost perform almost identically in this benchmark, both failing to match the performance of the 2-way TLS.

ep (Figure 5.3a) scales relatively well up to 16 cores, after which point there is a significant performance drop. The data partitioning is done statically, which leads to load balance problems. This is the case for the evaluated working set, which shows performance degradation that manifests as increased barrier time. The application has one hot loop which shows infrequent dependences and is amenable to speculative parallelization with synchronization around dependences. Moreover, we speculatively parallelize calls to *vranlc()* which precede this loop and account for the remaining execution time. This yields considerable performance improvement for the TLS versions. Both Core Fusion and Frequency Boost provide performance benefits, but are not able to match that of our scheme.

ft (Figure 5.3b) is a memory bound application that scales poorly. The memory bandwidth quickly becomes the bottleneck as the number of cores is increased,

[1]The rest of the serial time is spent on the *WriteGrayScaleTIFF()* function, a function heavy on I/O that we were unable to speculate upon.

and going beyond 8 cores offers no benefits. Speculatively parallelizing several loops helps improve scalability, but does not change the fact that the application is memory bound. Core Fusion performs relatively well by having a better long latency tolerance. Frequency Boost, on the other hand, exacerbates the memory boundedness issue and performs worse than the other schemes.

is (Figure 5.3c) exhibits poor scaling due to a coarse-grain critical section. We speculatively parallelize the critical section as well as the kernel preamble, which is executed only by the "master" (in *OpenMP* terminology) thread and is followed by a barrier. This provides an improvement in performance, enabling the application to continue scaling up to 16 cores for 2-way TLS and up to 64 cores for 4-way TLS. Both Core Fusion and Frequency Boost also improve performance, but only up to 16, and less so than 2-way in this case.

lu (Figure 5.3d) scales nicely up to 16 cores, but shows no scaling beyond this point. This is due to poor work partitioning which is done at the outermost level of the benchmark's loops that typically have an iteration count equal to a statically defined value (*ISIZ*). For the evaluated dataset this value is set to 12, thus any cores above this point do not receive work and spend most of their time in barriers. There is, however, an abundance of nested parallelism in several of the benchmark's loops and we speculatively parallelize inner loops in functions *rhs()*, *buts()*, *blts()*, and *jacu()*. The exploitation of this nested parallelism yields scalability improvement by the 2-way TLS, that is further enhanced with 4-way TLS. Core Fusion provides significant boost in performance – better than 2-way TLS – due to a high level of ILP that reflects in the high Instructions Per Cycle (IPC) that ranges from 1.12 to 1.67 on average for the Base version with 16 cores that utilizes the larger cores of Core Fusion. Frequency Boost performs in line with Core Fusion but for a different reason; the relatively low last-level cache misses of *lu* allows for almost linear performance boost with respect to frequency.

sp (Figure 5.3e) exhibits poor scaling beyond 16 cores and none beyond 32 cores mostly due to synchronization time spent in barriers due to load imbalance, and to a lesser extent due to memory boundedness. Most of the loops of this benchmark are nested ones with depths ranging from two to four and work partitioning is done at the outermost level, yielding work only for the first *PROBLEM_SIZE* (36 for the evaluated dataset) threads. We speculatively parallelize loops in functions *add()*, *lhsy()*, *lshz()*, *compute_rhs()*, *x_solve()*, *y_solve()*, and *z_solve()*, consisting mostly of inner loops of the aforementioned outer loops. This reduces the execution time of the outer loops,

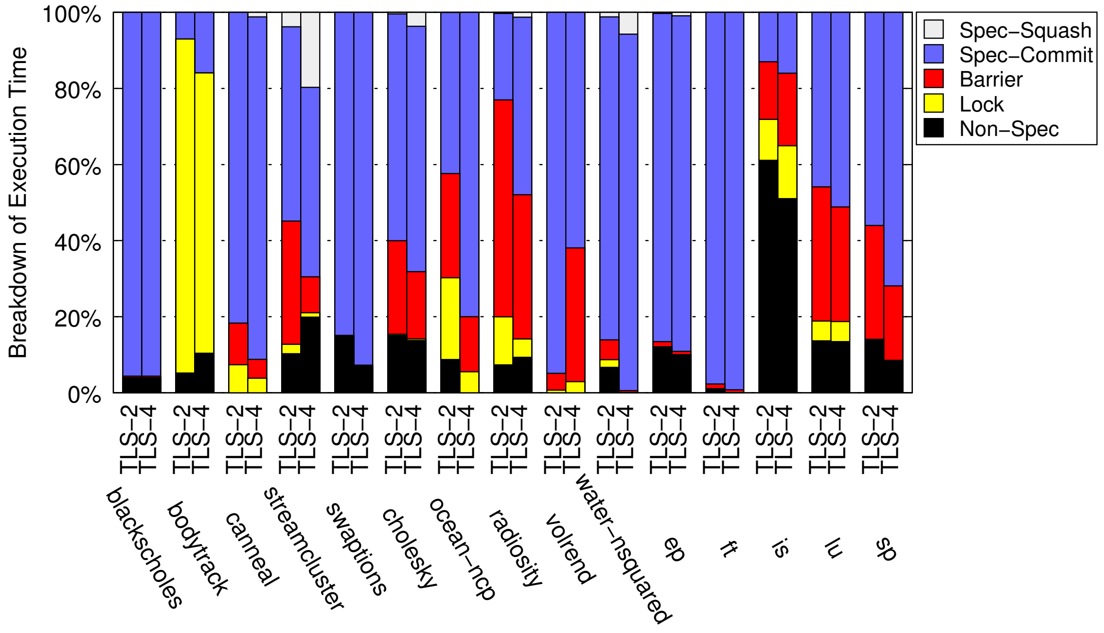


Figure 5.4: Breakdown of execution time for the best scalability point for each benchmark with 2-way and 4-way TLS, showing time spent in non-speculative regions, locks, barriers, and speculative regions. The speculative region is further partitioned into time spent in successful speculation and time spent squashing.

which in turn translates to less time spent in barriers. Core Fusion is able to provide some performance improvement while Frequency Boost fails to provide any noticeable benefits.

5.1.2 Summary

Overall, the results show that employing fine-grain implicit speculative threads to explicit threads beyond the scalability limit of the latter leads to performance gains in all the applications studied. In cases where there is enough fine-grain parallelism (e.g., loops with large trip counts) the 4-way TLS was able to improve on the 2-way TLS for even larger number of cores, after lagging behind for smaller number of cores. The alternative approaches of Core Fusion and Frequency Boost are also often able to provide performance gains, albeit not in all cases and not as high as the TLS schemes. The difference between TLS and Core Fusion shows that exploiting additional fine-grain TLP can be advantageous as compared to exploiting more ILP, specially when enough parallelism can be harvested for the 4-way TLS while fusing 4 cores becomes impractical. Finally, the results show that Frequency Boost can be effective on compute-bound applications, but is not as effective as the other approaches in memory-bound applications. A summary of these findings is depicted in Table 5.1.

Some of the exposed speculative parallelism exploited by TLS exhibited no data-dependences (e.g., *swaptions*) and could thus be exploited by a keener programmer. This entails, however, that the programmer must re-write *all* the applications in question, with each application requiring different approaches based on their algorithmic and data specifics. Our approach however is ubiquitous in the sense that it applies the same procedure to all the workloads (semi) automatically. Moreover, for the applications that do show real data dependences (e.g. *water-nsquared*) it is not clear whether the programmer could expose *any* additional safe parallelism.

5.2 Effect of Dataset Sizes

As shown in the previous section, the technique proposed in this thesis is effective in providing additional performance gains once applications stop benefiting from more explicit threads. So far we have assumed *strong*-like scaling by fixing the input dataset, i.e., the "Normal" dataset (see Table 4.3). A valid question is how the approach would fare under *weak*-like scaling conditions where the input of interest to the user is allowed to increase. To assess this we have also simulated the "Large" datasets, and present the results in Figures 5.5, 5.6,and 5.7.

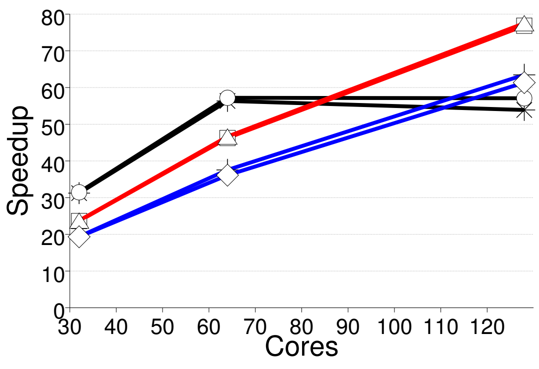
Overall, as expected, most applications achieve performance improvements with a larger number of cores with the larger datasets, one of them (*streamcluster*) showing good scalability up to the total system size. However, in several cases performance still stops scaling before the total number of cores available is reached. In these cases, we again observe that our approach with 2-way and 4-way TLS is able to provide further performance improvement with larger numbers of cores.

A more detailed analysis is as follows. *ep* (Figure 5.7a) and *swaptions* (Figure 5.5e) exhibit a shift in scalability of two and one points to the right, respectively, but the trends remain unchanged. Our scheme remains largely beneficial for these benchmarks. *lu* (Figure 5.7d) also exhibits a shift in scalability of one point to the right, with TLS being more beneficial for the larger dataset due to increased nested parallelism.

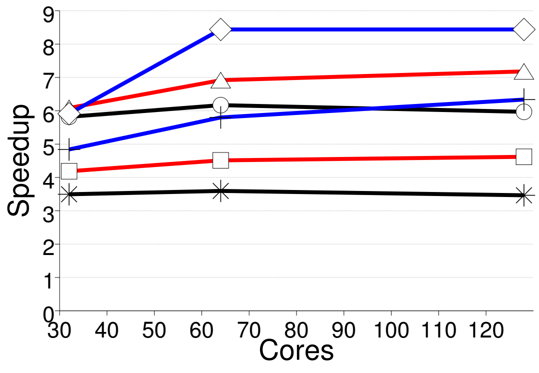
blackscholes (Figure 5.5a), *is* (Figure 5.7c), and *ft* (Figure 5.7b) are virtually unaffected by the change in the input data size. *bodytrack* (Figure 5.5b) exhibits the same trend in both datasets and no shift of scalability point, albeit shifted upwards in terms of speedup with the larger dataset. Similarly, *canneal* (Figure 5.5c), and *volrend*

Benchmark	**Base Scalability**	**Improvement Over Best Scalability**			
	Point (# cores)	2-way TLS	4-way TLS	Freq. Boost	Core Fusion
PARSEC					
blackscholes	64	36%	13%	19%	5%
bodytrack	32	28%	76%	25%	36%
canneal	32	23%	9%	7%	7%
streamcluster	64	13%	10%	7%	5%
swaptions	16	33%	79%	19%	20%
SPLASH2					
cholesky	32	13%	10%	9%	13%
ocean-ncp	32	20%	30%	6%	16%
radiosity	32	31%	33%	18%	13%
volrend	64	27%	42%	18%	8%
water-nsquared	64	34%	17%	16%	19%
NAS OpenMP					
ep	16	42%	98%	12%	5%
ft	16	85%	102%	13%	70%
is	8	31%	70%	14%	16%
lu	16	16%	23%	19%	21%
sp	32	28%	41%	5%	14%
avg		31%	44%	14%	18%

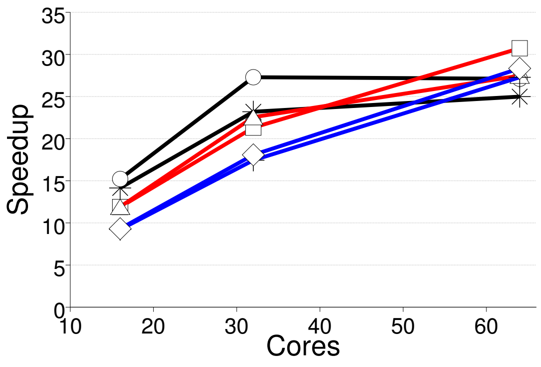
Table 5.1: Summary of the scalability behavior of the evaluated workloads along with the effect of 2-way and 4-way TLS, Core Fusion, and Frequency Boost.



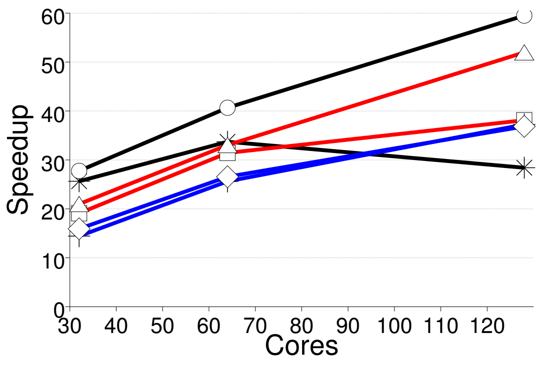
(a) blackscholes



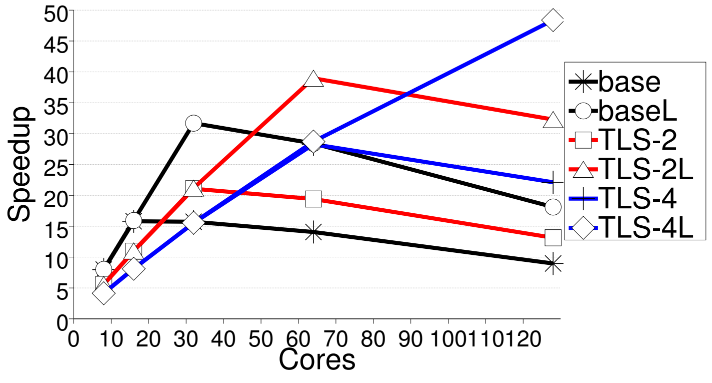
(b) bodytrack



(c) canneal

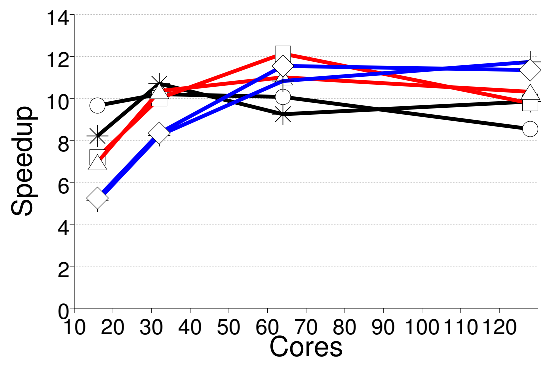


(d) streamcluster

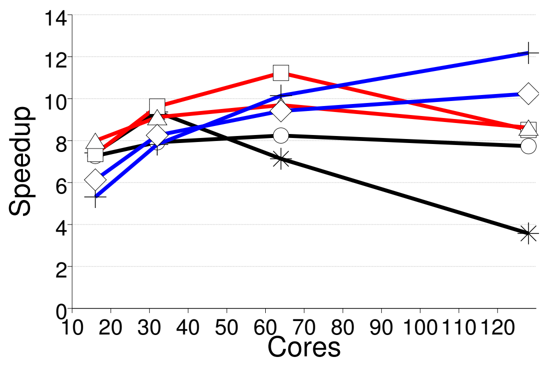


(e) swaptions

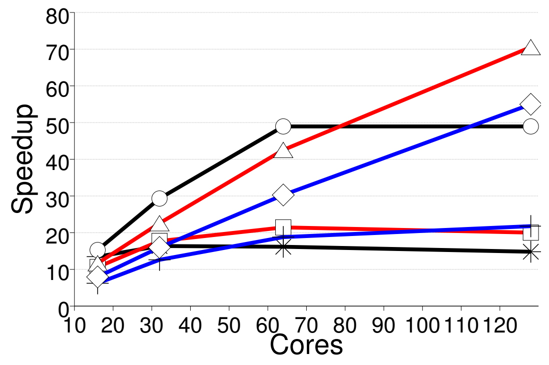
Figure 5.5: Performance and scalability of larger datasets. (part I: PARSEC)



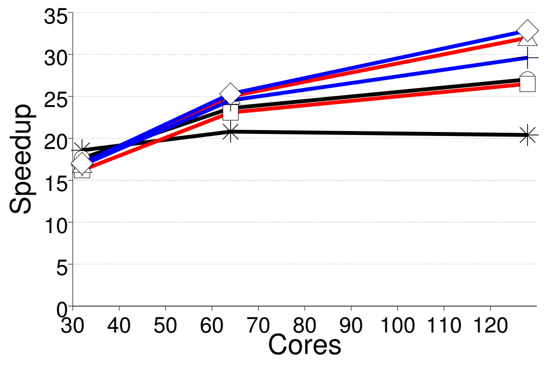
(a) cholesky



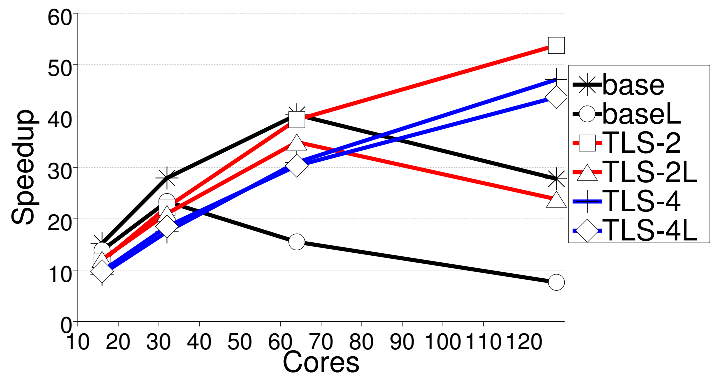
(b) ocean-ncp



(c) radiosity

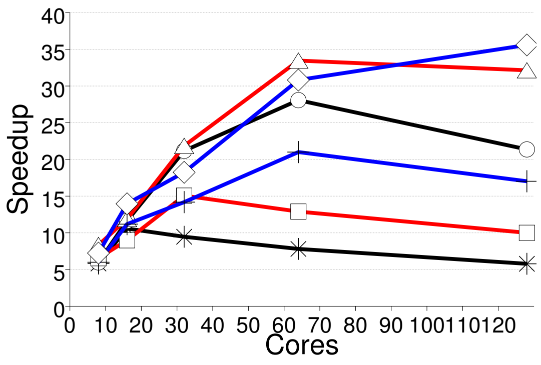


(d) volrend

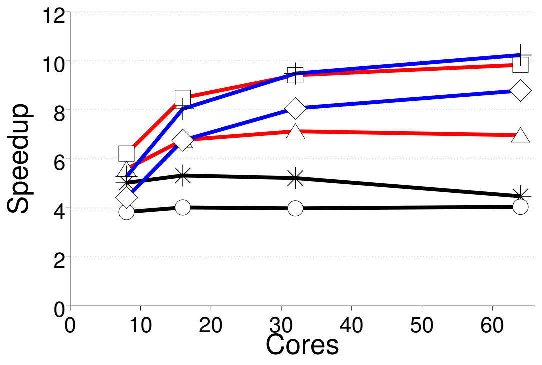


(e) water-nsquared

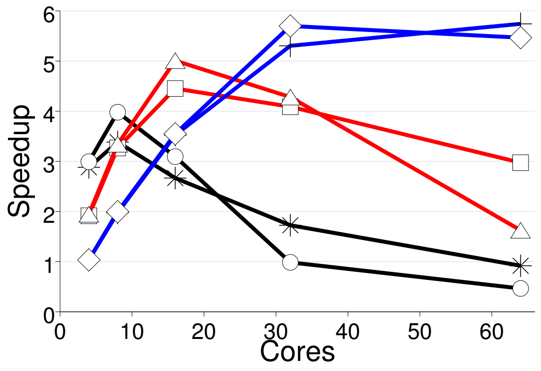
Figure 5.6: Performance and scalability of larger datasets. (part II: SPLASH)



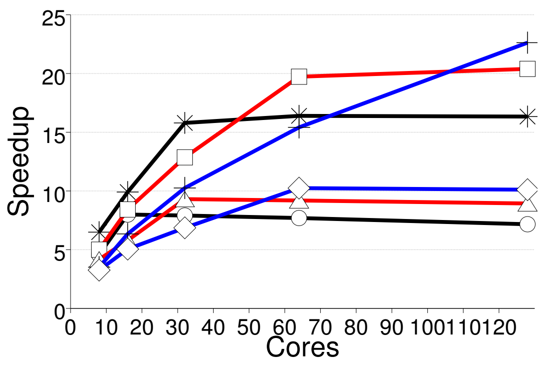
(a) ep



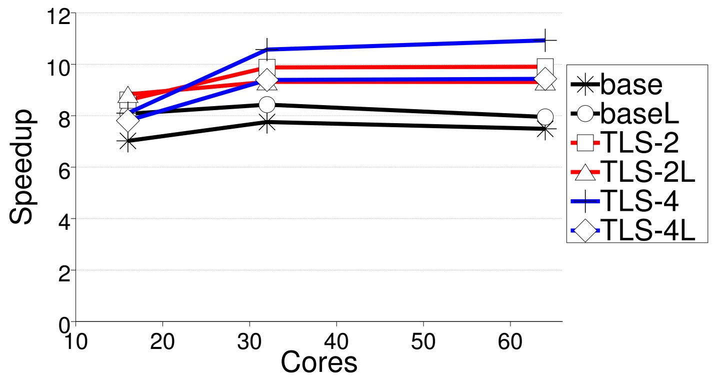
(b) ft



(c) is



(d) lu



(e) sp

Figure 5.7: Performance and scalability of larger datasets. (part III: NASPB)

(Figure 5.6d) exhibit a slight shift upwards of the scalability lines.

water-nsquared (Figure 5.6e), on the other hand, exhibits vastly different behavior between the two evaluated working sets. More specifically, the larger working set performs significantly worse than the normal one. This is due to the fact that the normal working set partitions evenly between the threads (both are a power of two), while the larger working set imposes load imbalance. Our scheme is effective in both cases, even more so for the larger dataset.

ocean-ncp (Figure 5.6b) exhibits a more "flat" behavior for the larger dataset which is largely due to it becoming memory bound. TLS remains beneficial, however, albeit by a smaller margin. *cholesky* (Figure 5.6a) also becomes more memory bound with the larger dataset which is reflected in the similar "flat" trend line.

radiosity (Figure 5.6c) exhibits significantly improved scalability with the larger dataset of the baseline version, showing almost linear scaling up to 64 cores. Speculation continues to enjoy considerable speedup on top of the baseline, however, enabling impressive scaling.

streamcluster (Figure 5.5d) shows improved scalability with the larger data that goes all the way to 128 cores, rendering the baseline the better performing option. The reduced gains of TLS are attributed to increased rates of mispeculation for the larger dataset.

5.3 Energy Consumption

Figure 5.8 depicts the total energy consumption of each of the evaluated schemes for all the benchmarks, normalized against the best scalability baseline point. If we take *is*, for example, the base case is for 8 cores, the 2-way TLS, Core Fusion and Frequency Boost are for 16 cores (and 8 explicit threads), and the 4-way TLS is for 64 cores (16 explicit threads). It is evident from this graph that Core Fusion's excessive power consumption is not coupled with enough performance improvement to be a viable solution energy-wise. *canneal* and *ft* are an exception to this, with Core Fusion more energy efficient than the baseline. In the case of *ft*, Core Fusion enjoys performance improvements that are on-par with the increased power consumption. In the case of *canneal* Core Fusion is able to attain its best performance using the *same* number of cores (i.e., half the number of explicit threads) as the best performing baseline. Also, the best performing baseline of 64 cores is only slightly faster than the 32 core one, while using significantly more power mostly spent on synchronization. Frequency Boost

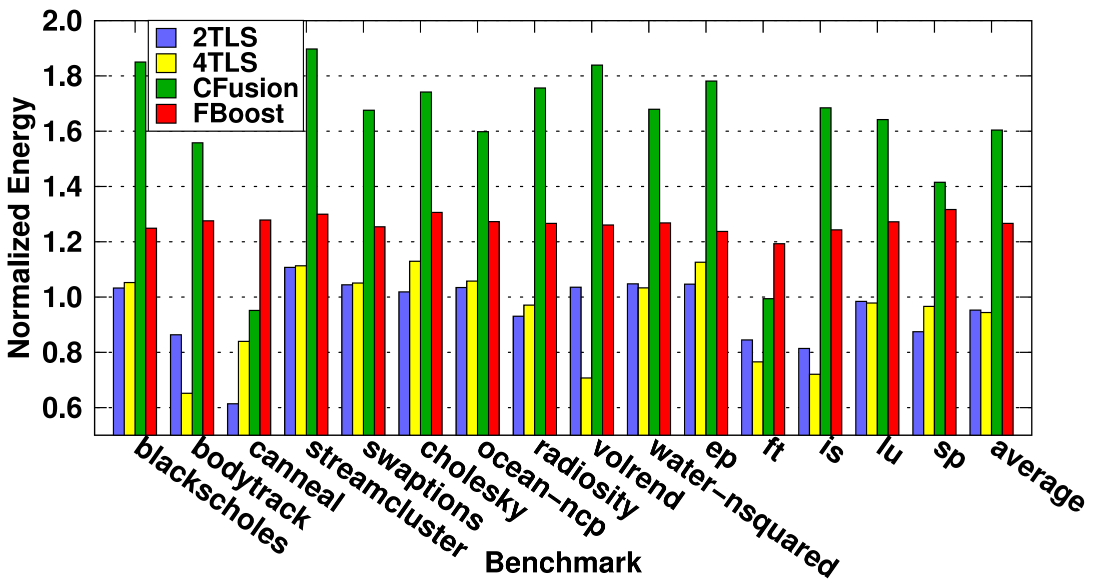


Figure 5.8: Energy consumption showing the best performing point for each scheme, normalized to the best performing base case.

offers better energy efficiency than Core Fusion overall but is still 27% more energy hungry than the baseline on average.

It is worth noting the variability of the number of fetched instructions for Core Fusion and Frequency Boost, as depicted in Figure 5.9. This is solely due to the variable execution of busy-wait synchronization primitives and is evident in the absence of variability for data parallel applications, like *blackscholes*, *swaptions*, and *ep*. Frequency Boost closely follows the baseline in terms of fetched instructions, except in the cases where the increased frequency magnifies load imbalance, thus amplifying the number of fetched instructions (*ocean-ncp*, *ft*, and *sp*). Core Fusion, on the other hand, reduces the number of fetched instructions for most workloads. This is due to reducing time spent on synchronization primitives for benchmarks with large serial and critical sections (*bodytrack*, *is*) or *OpenMP* applications with parallel regions with high ILP (*ft*, *lu*, and *sp*).

Our scheme is able to achieve significant performance improvements while maintaining a reasonable power envelope that yields total energy consumption close to – and sometimes better than – the baseline. Our scheme is less energy efficient than the baseline in cases of high mispeculation rates, like *cholesky*, or when it imposes increased contention of the shared resources, like *streamcluster* and, to a lesser ex-

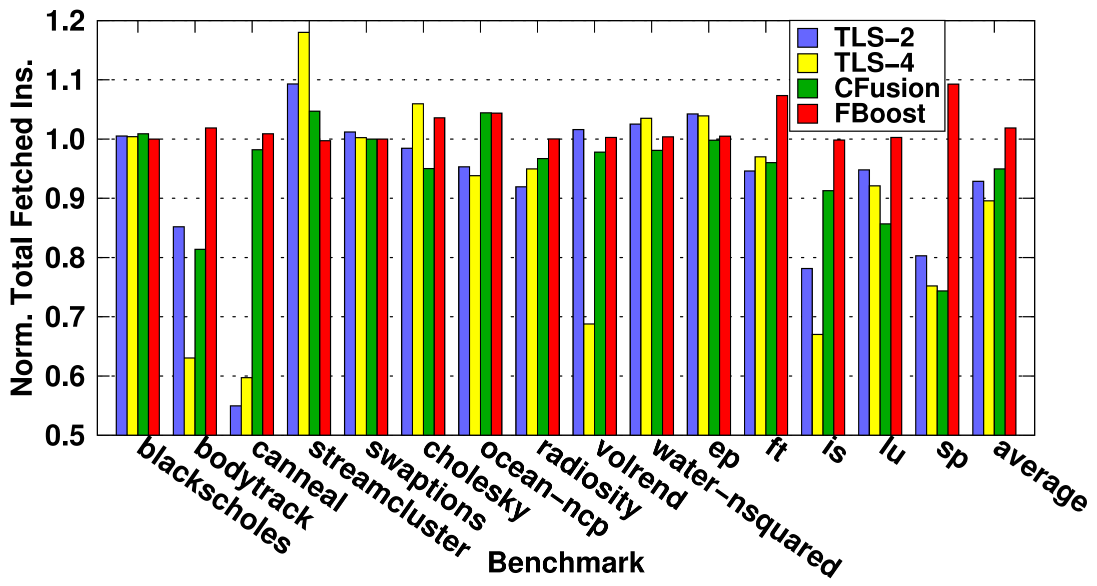


Figure 5.9: Aggregate sum of fetched instructions over all cores for the best performing point for each scheme, normalized to the fetched instructions of the baseline.

tend, 4-way *ep*. However, for *bodytrack*, *canneal* [2], *ft*, *is*, *sp*, 4-way TLS *volrend* [3], and to a lesser extend, *radiosity*, it is actually *more* energy efficient than the baseline. This is due to reduced time spent in the busy-wait synchronization primitives, which are particularly energy hungry. Moreover, this reduction in synchronization time is fully reflected in the total aggregate sum of fetched instruction over all the cores and is shown in Figure 5.9.

In the cases, however, with little to no synchronization overheads speculative parallelization is typically less efficient than the execution of explicit parallel threads This is true for *blackscholes*, *swaptions*, *ocean-ncp*, and *ep* with 2-way TLS. Furthermore, this comes as little surprise considering the extra hardware required to support speculation, the overheads in terms of extra instructions executed (depicted in Figure 5.9), and is in line with observations in prior work [60].

5.4 Dynamically Tuning the Number of Threads

As explained in Section 3.4 we have developed an auto-tuning mechanism to dynamically choose the number of explicit and implicit threads for *OpenMP* applications such

[2]We should note that for *canneal*, the best performing baseline is for 64 cores, the 2-way TLS and Core Fusion are for 64 cores, and the Frequency Boost and 4-way TLS are for 128 cores.

[3]The vastly reduced energy of the best 4-way TLS versus the 2-way is attributed to having half the number of explicit threads and thus spending significantly less time in barriers (Figure 5.2d).

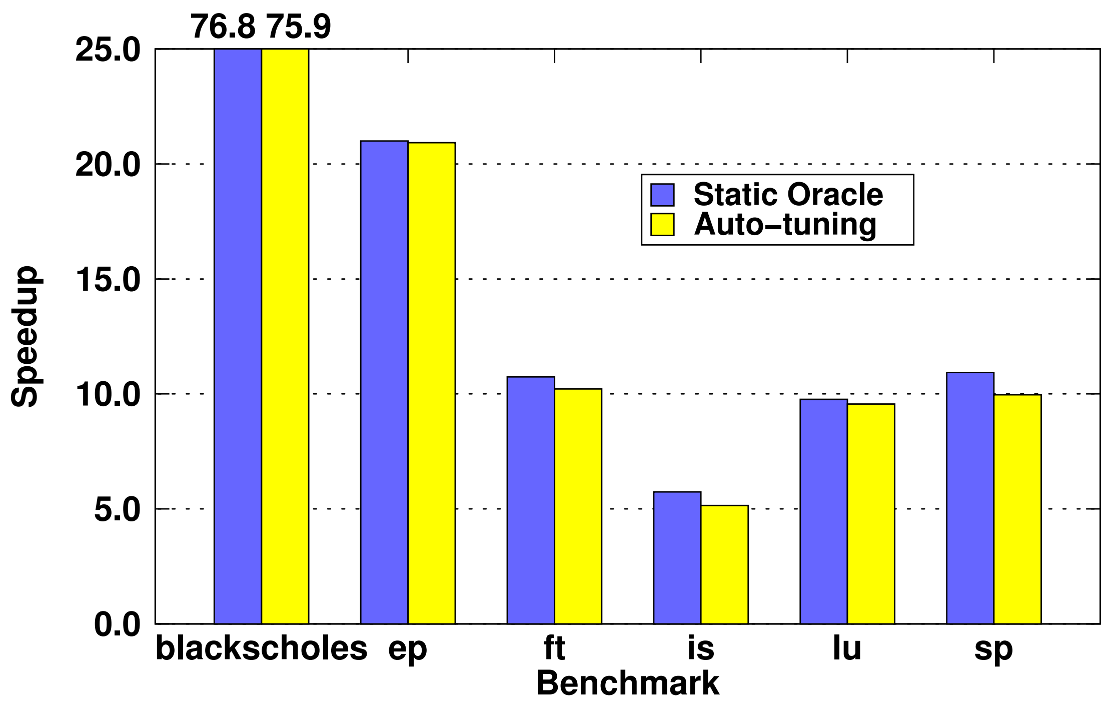


Figure 5.10: Dynamically tuning the number of threads.

that performance is maximized. We have employed this to all the evaluated *OpenMP* benchmarks apart from *bodytrack*, whose parallel regions include code dependent on the thread identifiers and does not support dynamically switching the number of parallel threads. Figure 5.10 compares this auto-tuning mechanism against the static optimal extracted from the results of Section 5.1. Our auto-tuning mechanism performs within 11% of the optimal in the worst case (*is*), less than 1% in the best case (*ep*), and 6% on average. *ep*'s high iteration count completely amortizes the training costs. The high number of iterations of the parallel region minimizes the training overhead for *lu* and *blackscholes* as well. In the case of *is*, however, the auto-tuning settles correctly at 8 explicit threads, which is the tipping point of the baseline, but not the optimal in performance when coupled with 4-way TLS (which is 16 explicit threads). It therefore settles for a sub-optimal thread count leading to notably less performance than the static oracle. *sp*'s parallel region is only encountered a few times and is thus more susceptible to training noise. *ft* also exhibits a low trip count parallel region, but shows less performance difference between adjacent thread counts. These results assert that the evaluated parallel applications retain their auto-tuning amenability even when we employ implicit threads. Furthermore, our proof-of-concept auto-tuning algorithm offers a feasible solution with performance benefits that are close to the static oracle.

Chapter 6

Adaptive, Hierarchical Power Management Scheme

This chapter discusses our power management scheme for multithreaded applications with implicit threads running on the shared-memory many-core multi-processor presented in Chapter 3 that aims at lowering total-chip energy consumption while maintaining performance. First, in Section 6.1, it discusses room for improvement in prior work on power management for multithreaded workloads as a motivation to our work. Second, in Section 6.2, it presents the adaptive phase-driven scheme that augments the nested speculative parallelism to make it more energy efficient and discusses the necessary hardware requirements to support our proposal.

6.1 Motivation

We identify some shortcomings in the prior work for power management on multithreaded workloads and try to amend them. More specifically, we focus on the proposed schemes from [9, 19, 69], as presented in Chapter 2. The thrifty barrier [69] uses the idleness at the barrier to move the faster cores to sleep by predicting barrier stall times based on prior behavior. Meeting points [19] assumes strict Single Program Multiple Data semantics (e.g., traditional OpenMP) and places “meeting points” at the end of each parallel loop. Each thread monitors its current progress based on these meeting points, compares it against the other threads’ progress and proportionally throttles down depending on how far ahead in execution it is compared with the slowest thread. Thread criticality predictors [9] (TCP) dynamically monitor the cache misses of each thread and characterize the threads suffering the most cache misses as "critical". Higher frequency and voltage is then applied to the critical threads; lower frequency and voltage pairs, proportional to the ratio of cache misses against the maximum cache misses, are applied to the remaining threads.

The identified shortcomings that leave room for improvement are the following:

- They assume that the parallel workload will be barrier intensive [9, 19, 69] and fail to show how they would deal with different types of parallel workloads. Moreover, "meeting points" [19] assumes strict Single Program Multiple Data (SPMD) semantics for the workload in addition to being barrier intensive. Other types of parallel workloads, like data-parallel, would not benefit from such schemes. In fact, we expect them to perform worse in those cases.

- Their power management is driven by predictors; they either try to directly predict the barrier stall time [69] and hence put threads to sleep or to predict the most "critical" thread(s) [9] and shift DVFS resources accordingly. Typically, the thread(s) that arrive late at a barrier are the ones that suffer the most cache misses. By running these thread(s) at increased frequencies it is assumed that it will lead to globally better execution time and power [9]. We see two problems to this: (a) running threads with high cache miss ratio at high frequencies is locally energy inefficient (as in [9]) and (b) possible misprediction would definitely lead to globally sub-optimal energy efficiency (as in [9, 69]).

- A limitation of their evaluation methodology is that they do not make comparisons with other static frequency settings apart from the highest frequency setting. They should instead try to improve the *best* static frequency setting in terms of the metric they try to optimize (e.g., Energy, Energy-Delay Product (EDP)). For example, we have found that running at lower frequency and voltage pairs can be *more* efficient in terms of total Energy consumption (see Section 7).

The behavior of each of the evaluated schemes in different types of parallel workloads is illustrated in Figure 6.1, in order to better understand their shortcomings. The Normal frequency and voltage pair is operating at 3.00GHz at 0.900V (Level 0 in Table 4.2), the Low frequency and voltage setting is operating at 2.40GHz at 0.800V (Level 2 in Table 4.2), and the Very Low frequency and voltage setting is operating at 1.80GHz at 0.700V (Level 2 in Table 4.2). First, in Figure 6.1a we show a barrier intensive SPMD parallel workload (e.g., the *sp OpenMP* benchmark from the NAS PB [7]) and assume perfect prediction accuracy. This ideal example depicts the theoretical behavior of these schemes under their target application space. The thrifty barrier (Figure 6.1a-ii) correctly puts threads that reach a barrier early to sleep and wakes them up just before the last thread reaches the barrier. The thread criticality predictors (Figure 6.1a-iii) correctly predicts the critical thread, and proportionally reduce the frequency of the other threads in order to minimize time spent in barriers. The meeting points (Figure 6.1a-iv) behaves identically to criticality predictors in this example.

Second, Figure 6.1b depicts the same parallel workload as in Figure 6.1a, but without assuming perfect prediction. The thrifty barrier (Figure 6.1b-ii) correctly predicts the thread behavior for the first barrier, but fails to do the same in the second one. Assuming that the behavior of the second barrier is predicted based on the first barrier (a realistic assumption under the thrifty barrier scheme in case both dynamic barriers are based on the same static barrier) the thrifty barrier will incorrectly wake up stalling threads early, resulting in wasted energy spent spinning full-speed at the barrier. The thread criticality predictor (Figure 6.1b-iii) first incorrectly predicts the thread running at core C0 as critical before correctly settling to the thread running at core C1, for the first barrier region. A similar situation is assumed at the second barrier region, switching between critical threads before settling correctly at the thread running at core C2. These mispredictions lead to an increase in execution time, as well as non-negligible time spent spinning at barriers at various frequency settings. Similarly, the meeting points (Figure 6.1b-iv) exhibits some energy inefficiency due to mispredictions.

Third, Figure 6.1c depicts a multithreaded workload with large serial sections and no barriers (e.g., the *bodytrack* benchmark from PARSEC [11]). The thrifty barrier (Figure 6.1c-ii) fails to perform *any* power management under a no barrier condition. The thread criticality predictors (Figure 6.1c-iii) incorrectly predicts the main thread (running at core C0) as non-critical since we assume that the previously idle threads (running at cores C1, C2, and C3) will show a higher cache miss ratio. This leads to an increase in total execution time. It will, however, have threads spinning at the synchronization points waiting for the serial sections to finish at the lowest frequency setting since we naturally assume cache misses only for the main thread (core C0) in those regions. The meeting points (Figure 6.1c-iv) show a small increase in execution time due to some training noise before settling to the actual critical threads in the parallel regions of execution. It fails, however, to perform *any* power management for the serial sections of the workload and the spinning threads are thus left running at the highest frequency.

Finally, Figure 6.1d depicts a data parallel workload with no synchronization (e.g., the *swaptions* benchmark from PARSEC [11]). The thrifty barrier (Figure 6.1d-ii) again fails to perform *any* power management. The thread criticality predictors' (Figure 6.1d-iii) assumption that running threads suffering the most cache misses at higher frequency and voltage pairs would lead to globally optimal energy consumption does not hold for data parallel workloads. Running memory-bound threads at high frequencies and compute-bound threads at lower frequencies is definitely *less* energy efficient in this scenario (akin to sequential and multi-programmed workloads [2, 5]). The meeting points also fails to adapt to this type of parallel workload – with no barriers and no strict SPMD semantics – exhibiting either random power management in case there are meeting points placed in the workload (as it is assumed in Figure 6.1d-iv) or no power management whatsoever in case there are none.

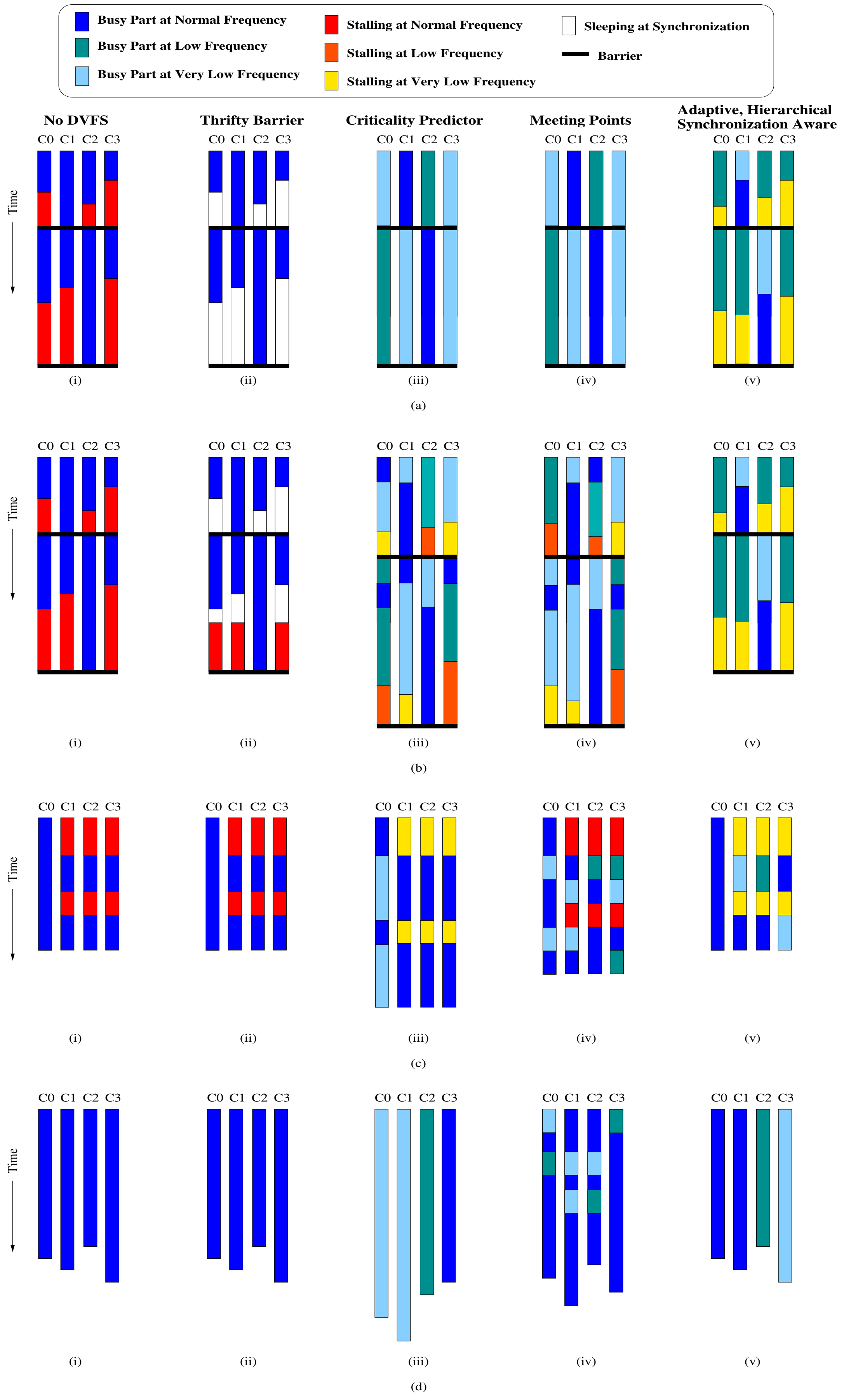
6.2 An Adaptive, Hierarchical Power Management Scheme

In order to overcome the above shortcomings, we propose a hierarchical power management scheme that is composed of two principal components: (a) a "local" component that tries to make a local optimal decision on a per-tile basis and (b) a "global" component that tries to make globally optimal decisions based on the synchronization behavior. The power management algorithm of our scheme is depicted in Algorithm 2. We describe the power management components in the following sections.

6.2.1 Local Component: Phase-Based Adaptive DVFS

The "local" component of our power management scheme is a DVFS controller based on a last-value phase-based predictor. Phase detection and prediction has been extensively studied in prior work [6, 33, 36, 98, 99]. The main purpose of phase characterization is to classify application execution into regions (or patterns) that show similar behavior. This phase detection and characterization can be done using various features, ranging from extremely fine grain features operating at a basic block level to larger features which typically involve metrics using performance monitor counters (e.g., instruction per cycles (IPC), misses per instructions (MPI)) sampled at large intervals. The granularity of the phase detection technique depends on its target application and the overheads of its operation.

Our goal is to apply phase detection and prediction within the scope of DVFS. In order to do so, we first have to identify a metric with which we are to classify differ-



Busy Part at Normal Frequency
Busy Part at Low Frequency
Busy Part at Very Low Frequency
Stalling at Normal Frequency
Stalling at Low Frequency
Stalling at Very Low Frequency
Sleeping at Synchronization
Barrier
No DVFS
Thrifty Barrier
Criticality Predictor
Meeting Points
Adaptive, Hierarchical Synchronization Aware
C0 C1 C2 C3
Time
(i)
(ii)
(iii)
(iv)
(v)
(a)
(b)
(c)
(d)

Figure 6.1: Behavior of the different schemes under different types of parallel workloads and conditions: (a) barrier-intensive workload assuming perfect prediction, (b) barrier-intensive workload with more realistic prediction accuracy, (c) parallel workload with large serial sections, and (d) data parallel workload.

ent phases and then build a phase-predictor. Since we will be changing the frequency dynamically, the metric that we choose has to be independent of frequency changes. Prior work [57] has shown that using *memory transactions per instruction* is a metric that (a) closely characterizes phase behavior and (b) is independent of frequency changes. Moreover, it obviously reflects the memory-boundness of each phase of execution [9, 32] and can be further used as direct input to the DVFS controller. We thus use cache misses per instruction (MPI) to both classify different phases and to drive frequency changes. To account for the greater impact in performance of last level cache (L2) misses compared to private cache misses we assign a larger weight to the last level cache misses. This weight is calculated as the ratio of last level cache latency to L1 miss latency [9]. The MPI value is then computed using the following equation:

$$MPI = (L1Misses + L2Misses \cdot (L2MissLat \div L1MissLat)) \div Ins \qquad (6.1)$$

Each tile periodically monitors the memory misses of its cores at every power management *quantum* (or interval) and performs DVFS control as depicted in lines 17-23 of Algorithm 2. The MPI value is computed at each quantum, mapped to a phase, which in turn is predicted to be the phase for the next quantum in a last-value fashion. The thresholds based on which the mapping between MPI value and voltage-frequency pair is chosen is presented in Table 6.1. Phases are essentially classified from compute-bound to completely memory bound and gradually mapped from the highest to the lowest frequency setting, respectively. These thresholds have been empirically chosen in an effort to minimize ED^2 and are thus biased towards higher frequencies. We have found that implicit threads incur higher contention to the shared resources and are naturally less dependent to frequency changes compared to the explicit thread only regions. This is reflected in the lower MPI values required for a frequency change in the case of implicit thread regions as opposed to explicit thread only regions as depicted in Table 6.1. Note that we place the power management module to the un-core portion of the tile and specifically between the L2 cache bank and the directory and router logic (Figure 6.2a) in order to (a) operate at a constant frequency (see Section 4.3), and (b) make the cache misses directly accessible to the module.

Frequent fluctuations in performance due to spurious program behavior can lead to wrong phase predictions and subsequently lead to sub-optimal choice of frequency settings. To amortize this effect we make use of a saturating counter table that contains a saturating counter for each of the voltage-frequency settings (Table 4.2). The values are initialized to zero for all the frequencies and to the maximum value for the highest

MPI range ETs	MPI range ITs	Phase #	Frequency (GHz)
[0.00 ,0.40]	[0.00 ,0.35]	0	3.00
(0.40, 0.80]	(0.35, 0.70]	1	2.70
(0.80, 1.20]	(0.70, 1.05]	2	2.40
(1.20, 1.60]	(1.05, 1.40]	3	2.10
(1.60, ∞)	(1.40, ∞)	4	1.80

Table 6.1: Phase classification and frequency mappings based on Misses Per Instruction (MPI) values, with different mappings for Explicit Threads(ETs) and Implicit Threads (ITs).

frequency. When the power manager decides on a new frequency for the next interval it first checks the saturating counter table's entry for that frequency. If it is at the maximum value, it proceeds with the frequency change. In any case, it increments the entry for the chosen frequency by one and decrements the other frequency entries by one (lines 25-29 of Algorithm 2).

6.2.2 Global Component: Synchronization Aware DVFS

We defer from increasing the frequency on low performing cores until we *know* that other cores are waiting for them to reach a barrier or release a lock/semaphore. To this end, it is essential to provide the local managers with information regarding the synchronization status of other threads in the system. We augment each tile with a *barrier_flag* and a *lock_flag* to indicate whether a tile is stalling on a barrier or lock. Global synchronization information is held in two shared bit vectors, a barrier stall bit vector and a lock stall bit vector. Each tile has write access only to the bit corresponding to its barrier stall bit and lock stall bit in the shared vectors. These shared bit vectors along with the accumulator logic that provide the number of tiles currently stalling in barrier and in locks are placed near the router logic of a centrally placed tile, as depicted in Figure 6.3a.

We annotate each barrier call in the program with a *start_barrier* instruction before the barrier call and a *end_barrier* after the call. Similarly, we annotate lock and semaphore (P semaphore calls only [106]) calls with *start_lock* and *end_lock* instructions. This annotation is done automatically in our source-to-source compilation framework (Chapter 4). Note that we support multiple different lock variables simultaneously as we only care whether we are within a barrier (lock) and not *which* particular barrier (lock) we are stalling on. Similarly sense reversing barriers [3] are fully supported since we annotate the calls to such barriers at the program calling site. Nested locks are not supported, however, as it did not affect the applicability of our scheme to the evaluated workloads [1].

During runtime the synchronization structures are updated dynamically. Upon encountering a *start_barrier* (*start_lock*) instruction the *barrier_flag* (*lock_flag*) is set, and a signal is sent to the shared barrier (lock) stall bit vector to set the bit corresponding to its tile. The flags and bits of the synchronization bit vectors are reset in a similar fashion upon encountering the *end_barrier* (*end_lock*) instruction.

Each tile polls the shared synchronization structures at a power management *quantum* granularity (same as the one used for the local managers) and obtains the number of threads currently stalling in a barrier or locks. Then, the power manager uses an empirical heuristic to choose an appropriate voltage-frequency setting (lines 12-16 of Algorithm 2). If the power manager on a tile detects that more than 20% of the explicit threads are stalling on a barrier or more than 80% of the explicit threads are awaiting to get a lock, then it characterizes itself as critical and thus raises its frequency to the highest value. Additionally, each tile power manager makes a local decision based on its synchronization status. If its barrier or lock flag is set it opts for the lowest possible voltage frequency setting (Level 4 in Table 4.2), as depicted in lines 10-11 of Algorithm 2.

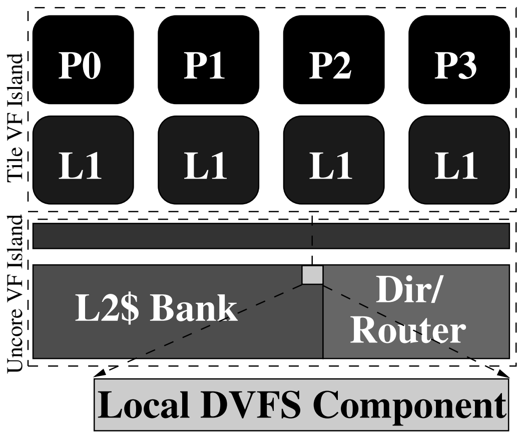
6.2.3 Expected behavior

The expected behavior of our power management scheme against the evaluated alternative schemes is depicted in Figure 6.1. Apart from the ideal case presented in Figure 6.1a, we expect our scheme to perform better than the alternatives. First, for barrier intensive workloads (Figure 6.1b-v), our scheme will operate as follows. For the parallel region and before reaching the barrier, all the threads will choose a frequency and voltage setting based on their cache miss behavior, with threads exhibiting high cache misses running at slower frequencies. Threads that reach the barrier will shift to the lowest possible frequency and voltage pair, with the rest of the threads increasing their frequency to the highest value to minimize energy wasted at the barrier. Second, a similar approach is taken for the parallel workload with large serial sections

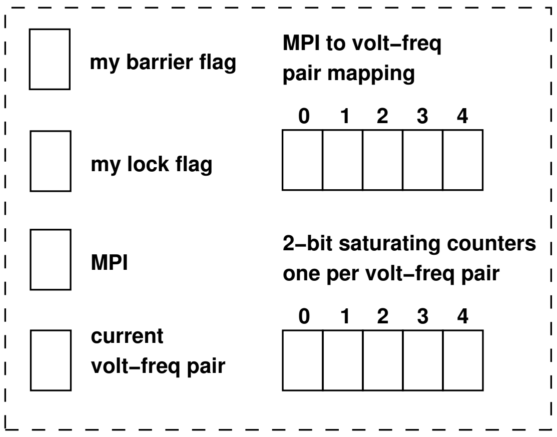
[1]Replacing the lock bits with dynamic stacks could trivially amend this shortcoming should this become necessary.

Algorithm 2 Local phase + Global sync power management

```
 1: // This is executed on each tile every 10K cycles
 2: // Step 1: Update synchronization flags and get current values
 3: $global\_barr\_array[myid] \leftarrow my\_barrier\_flag$
 4: $global\_lock\_array[myid] \leftarrow my\_lock\_flag$
 5: $barrier\_ratio \leftarrow barr\_stalled$
 6: $lock\_ratio \leftarrow lock\_stalled$
 7: // Step 2: Proceed to power management
 8: $volt\_freq \leftarrow curr\_volt\_freq$
 9: if $my\_barrier\_flag \vee my\_lock\_flag$ then
10:     // Either in barrier or lock, opt for lowest frequency
11:     $volt\_freq \leftarrow VOLTFREQ[4]$
12: else if $(barrier\_ratio \geq 0.2) \vee (lock\_ratio \geq 0.8)$ then
13:     // More than 20% of the threads are stalling on a barrier or
14:     // more than 80% of the threads are waiting to get a lock,
15:     // then choose the highest frequency
16:     $volt\_freq \leftarrow VOLTFREQ[0]$
17: else
18:     // Synchronization information yielded none of the other heuristics,
19:     // proceed to MPI-based last-value based prediction and DVFS
20:     $MPI \leftarrow (l1Misses + ((l2Misses \cdot l2MissLat) \div l1MissLat)) \div Ins$
21:     $quantise\ MPI\ based\ on\ empirical\ threshold$
22:     $volt\_freq \leftarrow VOLTFREQ[MPI\_QUANTIZED]$
23: end if
24: // Step 3: Update saturating counters and apply DVFS settings
25: if $(SATCNT[volt\_freq] = MAX) \wedge (volt\_freq \neq current\_volt\_freq)$ then
26:     $set\ voltage\ and\ frequency\ to\ new\ setting$
27: end if
28: $increment\ saturating\ counter\ for\ selected\ volt\_freq\ by\ 1$
29: $decrement\ saturating\ counters\ for\ other\ voltage\ and\ frequency\ pairs\ by\ 1$
```



(a)


my barrier flag
my lock flag
MPI
current volt–freq pair
MPI to volt–freq pair mapping
0 1 2 3 4
2–bit saturating counters one per volt–freq pair
0 1 2 3 4

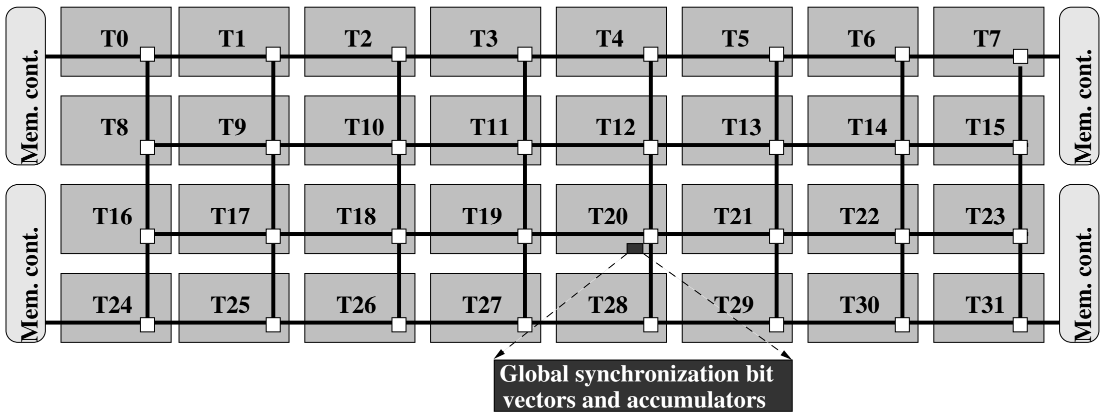
(b)

Figure 6.2: Local power management component: (a) Placement of the local power management component inside a tile, and (b) the hardware structures required per tile for the local power management component includes the synchronization flags, MPI and current voltage-frequency (VF) pair registers, the thresholds for MPI to VF mappings, and 2-bit saturating counters for each of the VF pairs.

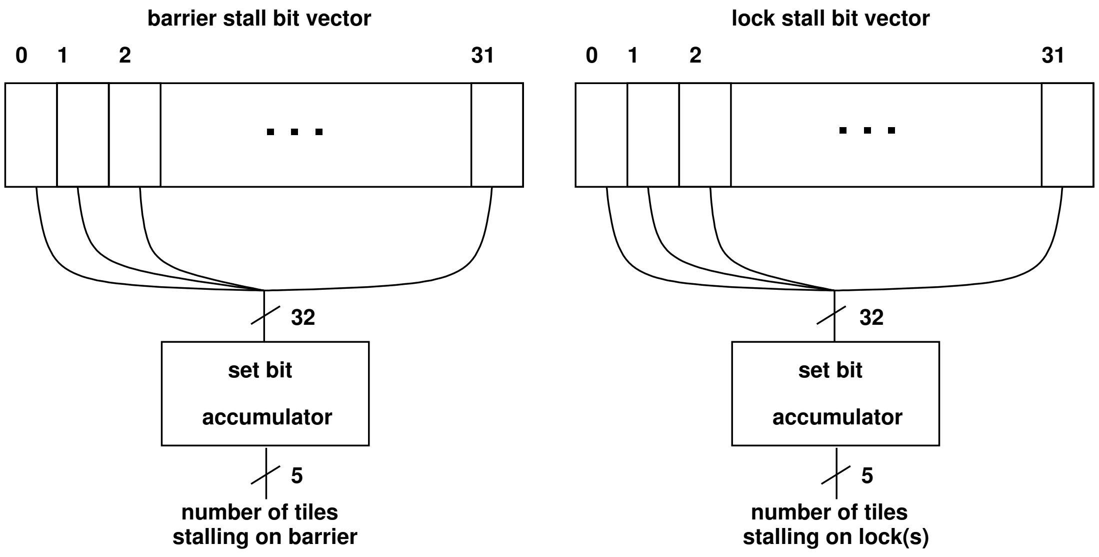
(Figure 6.1c-v). During the serial section, the main thread knows that other threads are waiting for it to finish and thus runs at the highest possible frequency, with the remaining threads waiting on the synchronization points at the lowest frequency and power setting. In the parallel section, each thread chooses a frequency and voltage setting that locally maximizes its energy efficiency. Finally, our scheme is able to handle data parallel workloads as well (Figure 6.1d-v). The local components will remain active throughout execution choosing appropriate frequency and voltage pairs depending on each thread's memory behavior.

6.2.4 Hardware Support

The power management algorithm detailed in Algorithm 2 highlights the hardware augmentations required to support our scheme. First, the local component assumes the additional hardware structures illustrated in Figure 6.2b: a 1-bit *my_barrier_flag*, a 1-bit *my_lock_flag*, a 3-bit register to store the quantized misses per instruction value, a 3-bit register to store the current voltage-frequency pair, five 2-bit saturating counters, one for each voltage-frequency pair, and two 1-bit flags to store whether the number of cores stalling on a barrier or locks are above a certain threshold. Second, the global component assumes the additional hardware structures illustrated in Figure 6.3b: a



(a)



(b)

Figure 6.3: Global power management component: (a) Placement of global synchronization bit vectors and accumulators, and (b) control logic for shared barrier stall and lock stall bit vectors along with accumulators that provide the number of tiles currently stalling in barrier and in locks.

32-bit barrier stall bit vector (i.e., one bit per tile), a 32-bit lock stall bit vector, and two 5-bit registers storing the current number of stalled tiles in barrier or in locks, respectively. The total number of additional hardware storage required is a mere 640 bits (80 bytes) for all the local components (20 bits each) and 74 bits ($<$ 10 bytes) for the global component.

Chapter 7

Analysis of the Adaptive, Hierarchical Power Management Scheme

This chapter evaluates the proposed hierarchical power management scheme. First, in Section 7.1, it discusses the amenability of the implicit threads to power management schemes. Second, in Section 7.2, it presents the bottom line results comparing this scheme against the current state-of-the-art power management schemes for the evaluated workloads with implicit threads, and it further performs a detailed analysis on the efficacy of our proposal. Finally, in Section 7.3, it discusses the effectiveness of the proposed power management scheme under explicit threads only.

7.1 Effectiveness Under Implicit Threads

In this section, we briefly evaluate the impact of implicit threads in the behavior of some of the power management schemes for multithreaded workloads discussed in Chapter 4 [9, 19, 69]. In order to do so, we first implemented these schemes in our simulator infrastructure (Chapter 4) and simulated the best scalability point for each workload (Chapter 5), with and without implicit threads. Details for the DVFS simulation and for the implementation of the evaluated schemes can be found in Section 4.3.

The metric we evaluate is Energy Delay^2 (ED^2), which puts more emphasis on performance than energy, as our aim is not to hurt performance [93]:

$$ED^2 = Power \cdot Time^3 \tag{7.1}$$

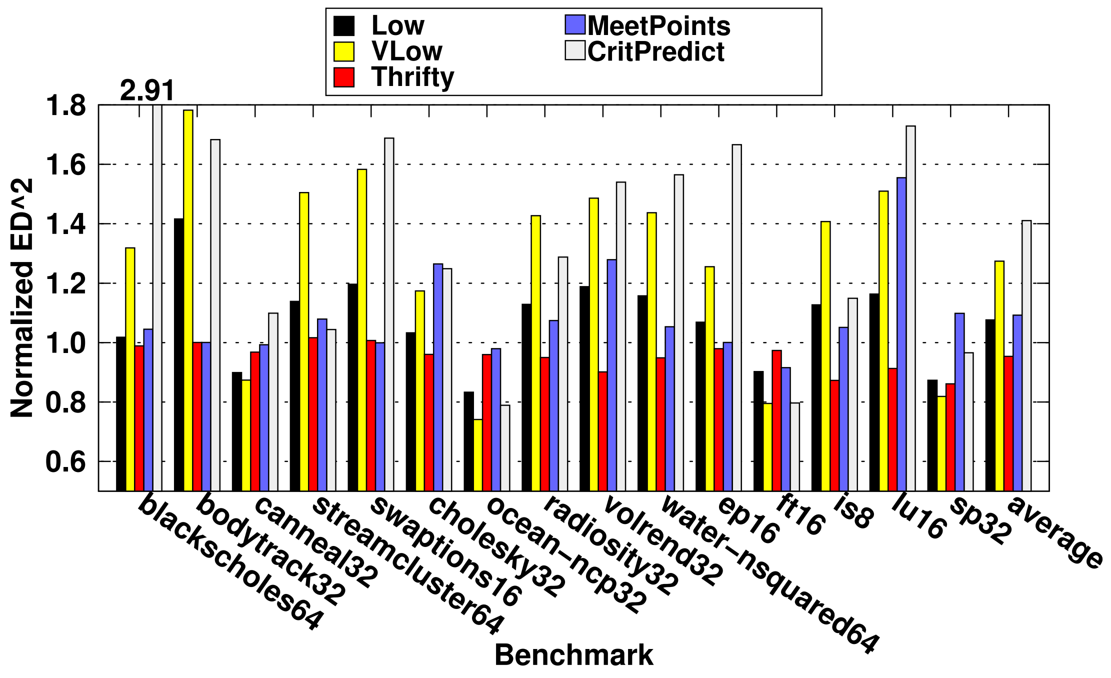
The results are depicted in Figures 7.1a and 7.1b running explicit threads only and explicit plus implicit threads respectively. The results show the best scalability point for each benchmark, based on the results presented in Chapter 5. The ED^2 bars are normalized to a baseline with no power management, running constantly at the highest voltage-frequency setting of 3.00GHz at 0.900V (Level 0 in Table 4.2). For each set of bars, from left to right, we see the ED^2 performance for running at a constant Low voltage-frequency setting of 2.40GHz at 0.800V (Level 2 in Table 4.2), running at a constant VLow voltage-frequency setting of 1.80GHz at 0.700V (Level 4 in Table 4.2), the thrifty barrier [69] power management ("Thrifty"), meeting points [19] power management ("MeetPoints"), and the thread criticality predictors [9] power management ("CritPred"). The take-away from these two graphs is that implicit threads do not affect the amenability of a multithreaded application to DVFS management. Applying any of the evaluated alternative power managements schemes on top of explicit threads only or explicit plus implicit threads yields almost identical results.

7.2 Bottom Line Results

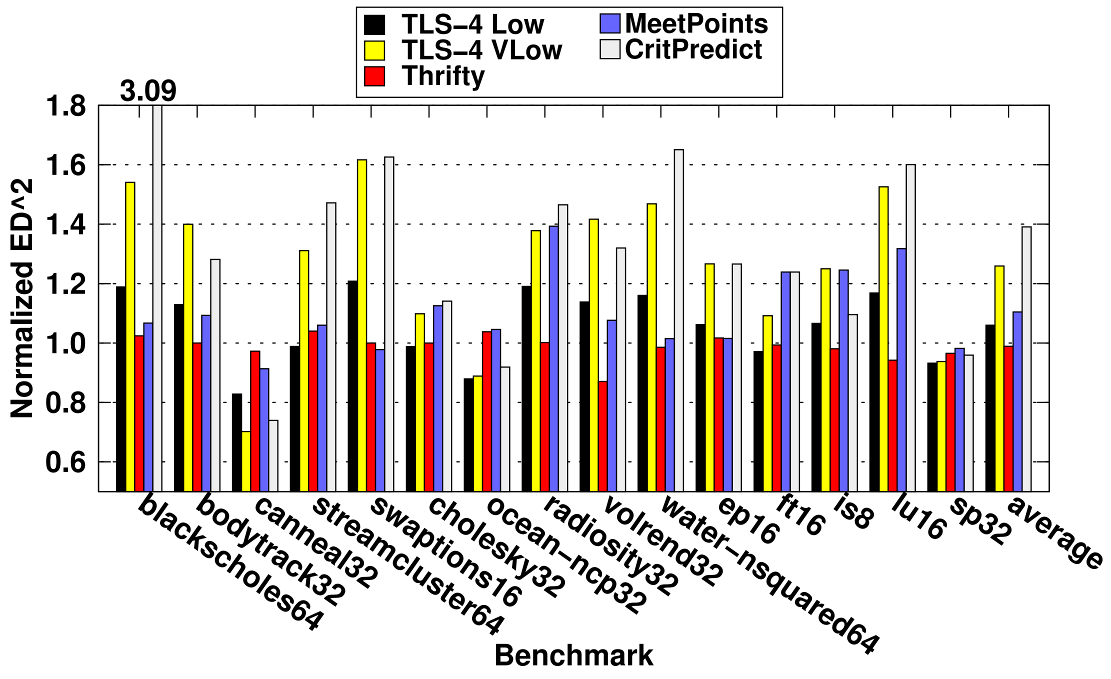
The bottom line results showing the performance of our power management scheme are illustrated in Figures 7.2, 7.3, and 7.4, showing Energy Delay^2 product, execution time, and power, respectively. All these results are using the best performing scalability point for each workload *including* implicit threads and are normalized against having no power management. The number of explicit threads is depicted with each benchmark. For each set of bars, from left to right, we see the performance of (a) running at a constant Low voltage-frequency setting of 2.40GHz at 0.800V (Level 2 in Table 4.2), (b) running at a constant VLow voltage-frequency setting of 1.80GHz at 0.700V (Level 4 in Table 4.2), (c) the thrifty barrier [69] power management ("Thrifty"), (d) meeting points [19] power management ("MeetPoints"), (e) the thread criticality predictors [9] power management ("CritPred"), and (f) our MPI adaptive plus synchronization-aware scheme ("MPI_Adapt_Sync").

Our scheme is able to significantly outperform both the baseline and the best performing alternative scheme (thrifty barrier) in terms of ED^2 (Figure 7.2). The other two alternative schemes compared (meeting points and thread criticality predictors) perform worse than the baseline in terms of ED^2. We first analyze the performance of the alternative schemes. The thrifty barrier is only able to marginally outperform the baseline. This is due to three reasons: (a) our best performing baseline being more powerful and energy efficient than the one used in [69] [1], (b) using lower DVFS in-

[1]The baseline used in [69] does not constitute the best scalability point for each evaluated workload,



(a) Explicit threads only



(b) Explicit and implicit threads

Figure 7.1: Energy Delay^2 product for the three prior power management scheme evaluated, normalized to a baseline with no power management. (a) Explicit threads only and (b) Explicit and Implicit threads. The number for each benchmark represents the number of explicit threads.

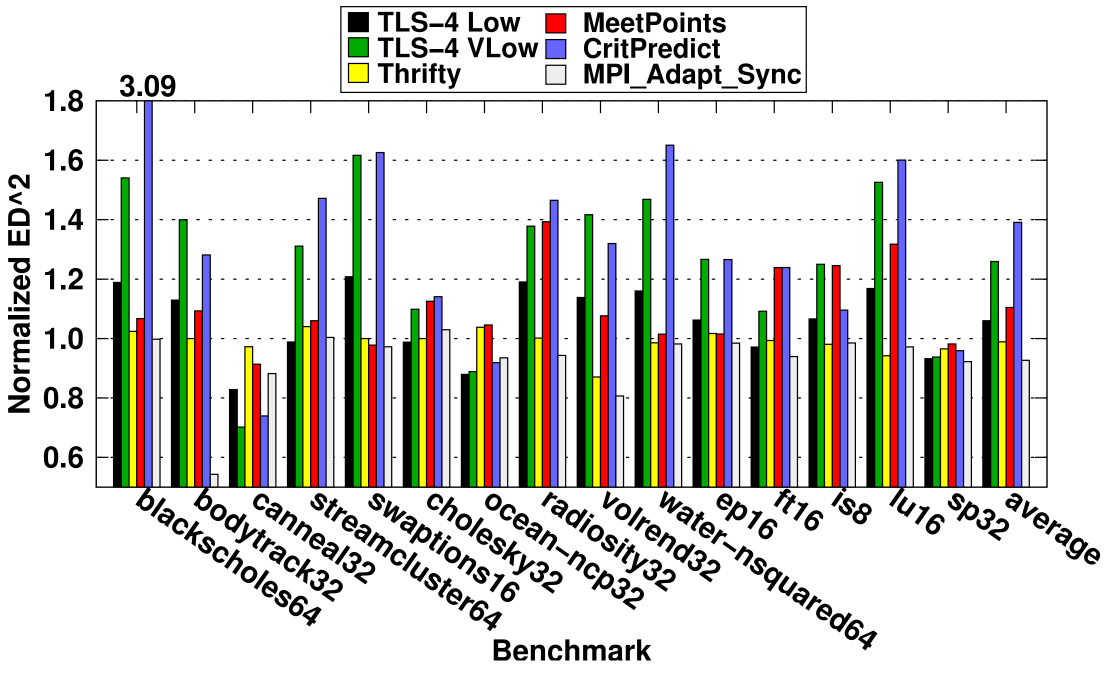


Figure 7.2: Energy Delay^2 product for each power management scheme, normalized to the best scalability point with implicit threads with no power management.

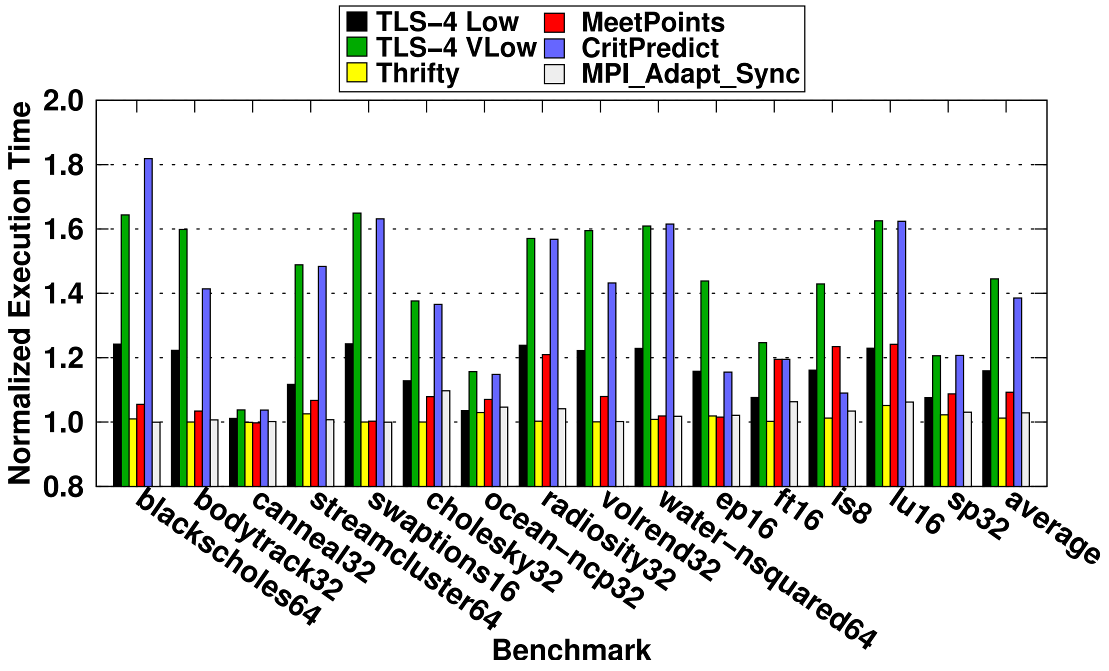


Figure 7.3: Execution time for each power management scheme, normalized to the best scalability point with implicit threads with no power management.

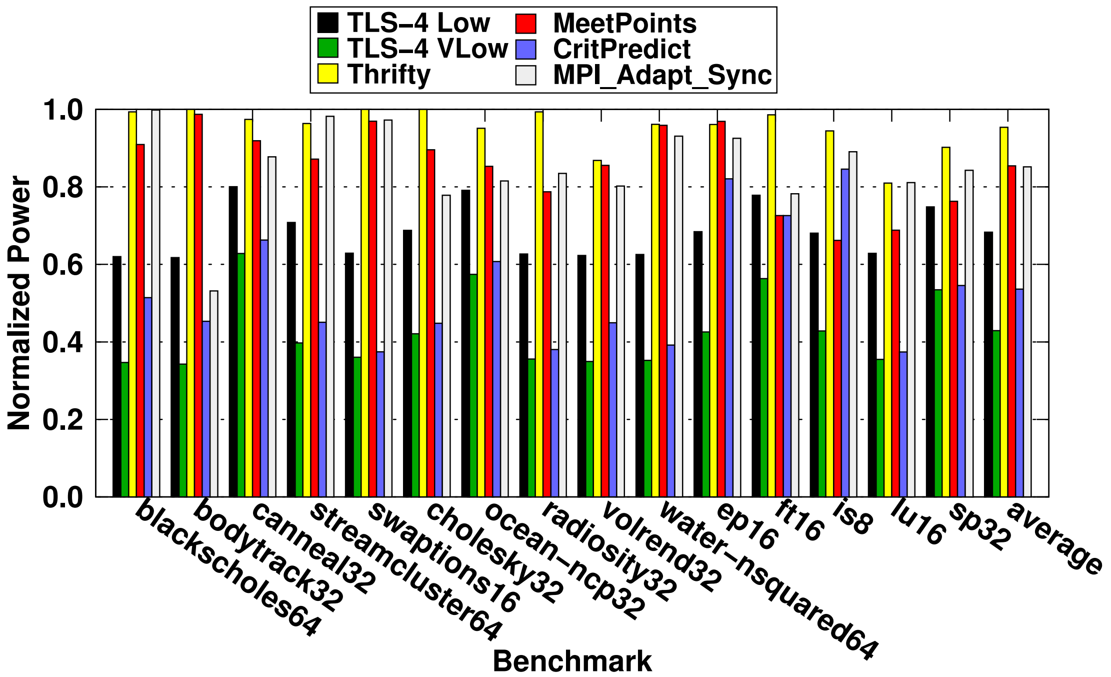


Figure 7.4: Average power consumption for each power management scheme, normalized to the best scalability point with implicit threads with no power management.

stead of going to sleep, and (c) the non-applicability of the scheme to applications with little or no barrier time (*blackscholes*, *swaptions*, *bodytrack*, *cholesky*, *radiosity*, *ep*, and *ft*). We attribute the performance of thread criticality predictors to the following causes: (a) the results shown in [9] are for in-order cores with blocking caches while in this study we show results for aggressive out-of-order superscalar cores with non-blocking caches, (b) we include data-parallel workloads in our evaluation for which this scheme was not intended (e.g., *blackscholes*, *swaptions*), (c) we aim at minimizing ED^2, while their aim was more towards energy and Energy Delay Product (EDP), for which thread criticality is still beneficial compared to the baseline. The meeting points scheme performs poorly in terms of ED^2 due to partially similar reasons: (a) having several workloads without strict SPMD semantics (e.g., *bodytrack*, *cholesky*, *radiosity*), (b) simulating aggressive 4-issue out-of-order cores with non-blocking caches as opposed to in-order cores with blocking caches used in [19], and (c) the difference in the implementation of meeting points compared to the original proposal in [19] as discussed in Section 4.3.

On the other hand, our scheme enjoys a significant reduction in terms of Energy

artificially leaving more room for improvement.

Delay^2 product of as much as 46% (*bodytrack)*, and 8% on average, compared to the baseline as is depicted in Figure 7.2. This is due to a reduction in power consumption of as much as 47% (*bodytrack*), and 15% on average, (Figure 7.4) with a minimal loss in performance of less than 3% on average and 9% in the worst case for *cholesky* (Figure 7.3). The performance of our scheme can be attributed to the following characteristics: (a) it is able to accurately adapt to the performance of each thread independently of synchronization by utilizing the misses per instruction at a local level, (b) it lowers the frequency of threads that stall on synchronization, and (c) it boosts the performance of cores when it is certain that they are critical for performance since other threads are waiting for them at synchronization points. These three characteristics make our scheme applicable to *all* the evaluated workloads – data-parallel, barrier-intensive, and workloads with large critical sections.

Figure 7.5 shows the breakdown of execution time into busy and synchronization (divided into barrier time and lock time) for all the power management schemes, normalized against the best scalability point of the baseline with implicit threads and no power management, in order to better assess the difference in behavior between the different power management schemes. First, we observe that the static schemes (4-way TLS running at low and very low voltage frequency pairs) naturally exhibit increased execution time compared to the baseline, except for memory bound benchmarks with very small serial sections (*canneal*, and *ocean-ncp*). Compute-bound workloads (*blackscholes*, *swaptions*, *radiosity*, *ep*, and *lu*) show a big drop in performance without, however, exhibiting a change in the synchronization to busy ratio. Workloads with large sequential parts of execution where threads stall waiting the main thread to finish, exhibit an increase in their synchronization to busy ratio (*bodytrack*, and *volrend*). Second, the thrifty barrier incurs minimal change in behavior compared to the baseline, as it only slows down threads only when they are stalling in barriers and does not affect the busy part of the multithreaded execution. Third, the meeting points has a minimal effect in benchmarks with just a few barrier calls (*bodytrack*, *cholesky*, and *swaptions*), shows little increase in execution time for homogeneous workloads (*blackscholes*, *canneal*, *streamcluster*, *cholesky*, *water-nsquared*, *ep*, and *sp*), and adversely affects the performance of imbalanced workloads (*radiosity*, *volrend*, *ft*, and *lu*). Fourth, the criticality predictors power management scheme significantly increases the execution time of data-parallel workloads (*blackscholes*, *swaptions*, and *ep*), of workloads with large sequential parts of execution (*bodytrack*, and *volrend*), and of compute bound workloads with few cache misses (*radiosity*, *water-nsquared*, and *lu*).

It performs favorably both in terms of execution time and in terms of ED^2, however, in the case of memory bound workloads with frequent cache misses (*canneal*, *ocean-ncp*, and *sp*). Finally, our scheme, on the other hand, performs closely to the baseline both in terms of execution time and in terms of busy to synchronization ratio across benchmarks. The only exception to this are *cholesky* and *lu*, both of which are very sensitive to frequency changes. We analyze the performance of our scheme in depth in the next paragraphs.

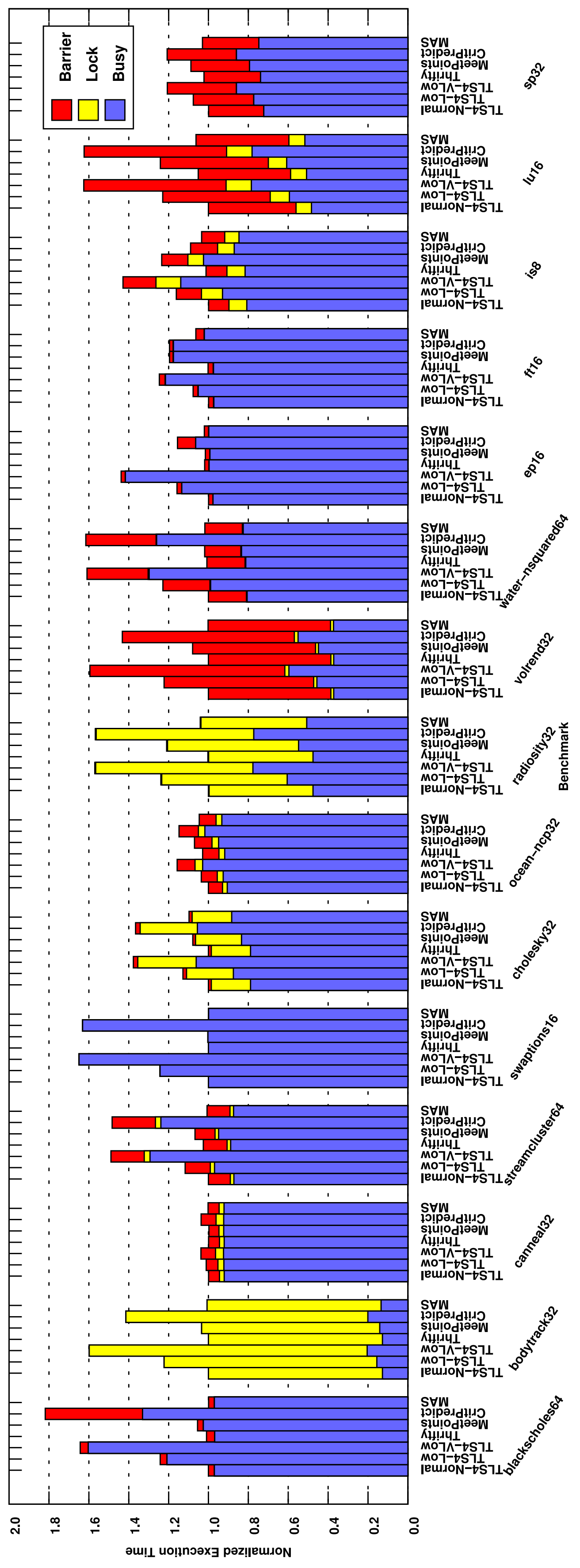


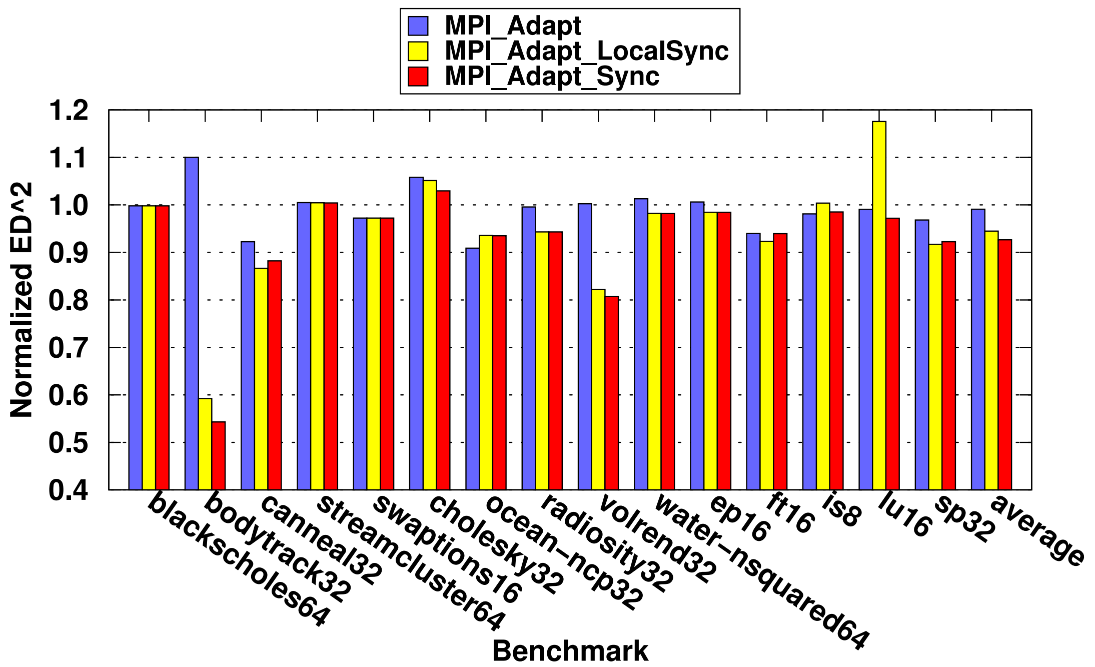
Figure 7.5: Breakdown of normalized execution time to busy versus synchronization (further divided into barrier and lock) using the different power management schemes with implicit threads. “TLS4-Normal”: 4-way TLS at normal voltage-frequency pair (i.e., the baseline), “TLS4-Low”: 4-way TLS at low voltage-frequency pair, “TLS4-VLow”: 4-way TLS at very low voltage-frequency pair, “Thrifty”: Thrifty barrier power management, “MeetPoints”: Meeting points power management, “CritPred”: Criticality predictors power management, “MAS”: our MPI adaptive plus synchronization-aware scheme.

7.2.1 Detailed Analysis

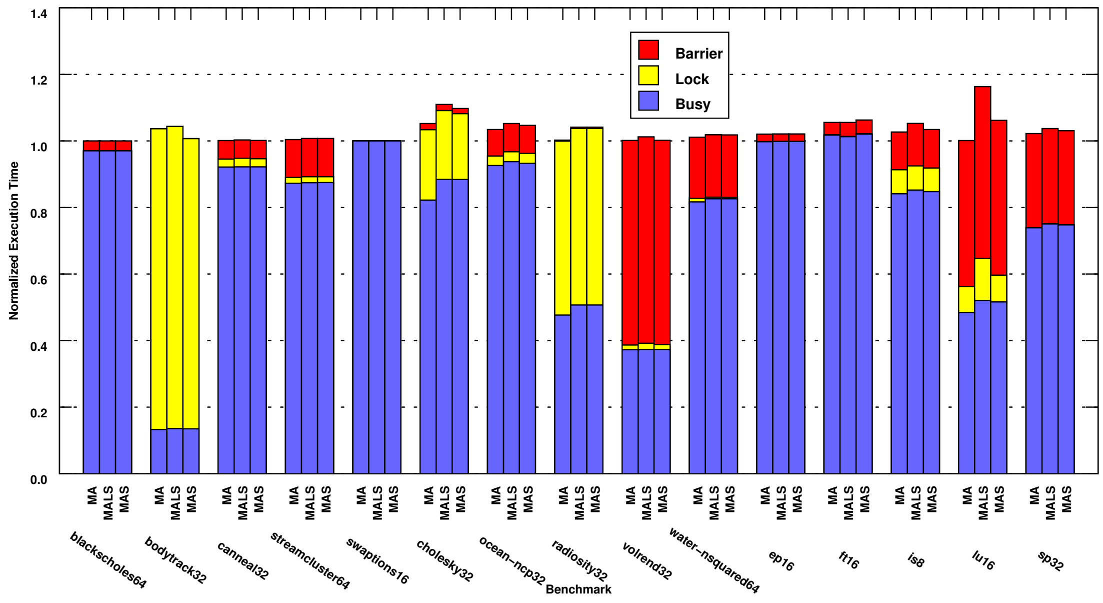
We first analyze the impact of the synchronization component of our scheme as presented in Section 6.2.2 and then analyze the effect of using the saturating counter table to mitigate fluctuations in performance.

7.2.1.1 Synchronization Aware Power Management

Figure 7.6a illustrates the performance in terms of ED^2 of the different components of our power management scheme: (i) using the local MPI-driven DVFS only ("MPI_Adapt"), (ii) adding local synchronization-aware DVFS module ("MPI_Adapt_LocalSync"), and (iii) adding the global synchronization-aware module ("MPI_Adapt_Sync"). This plot clearly depicts the additive gains of our scheme components. First, the MPI-adaptive component is able to deploy power savings at minimal performance degradation for benchmarks that exhibit memory bound phases, like *canneal*, *ocean-ncp*, *ft*, and to a lesser extend, *sp*, *is*, and *swaptions* (Figures 7.6a and 7.6b). The only benchmarks for which it shows poor ED^2 performance are *bodytrack*, and *cholesky*. This is due to slowing down critical threads. Second, the local synchronization optimization where each threads lower its frequency when stalled at a synchronization point significantly lowers the power consumption of workloads that exhibit long synchronization stalls. This is true for *bodytrack*, *volrend*, *canneal*, *sp*, *water-nsquared*, and *ep*. By not taking into account global synchronization behavior, however, it adversely affects the performance of benchmarks with either frequent synchronization calls, like *ocean-ncp*, *lu*, *cholesky*, and *radiosity*, or with critical threads, like *lu*, *ocean-ncp*, *cholesky*, *bodytrack*, *is*, *bodytrack*, and *water-nsquared* (Figure 7.6b). The final optimization that takes into account global synchronization information is able to mitigate this performance degradation. By boosting critical threads when other threads are waiting for them we are able to regain performance lost from the local synchronization power optimization in *lu*, *is*, *ocean-ncp*, *cholesky*, *bodytrack*, *is*, *water-nsquared*, and *volrend*. This is directly reflected in reduced time spent in synchronization primitives when going from the MPI adaptive plus local optimization scheme ("MALS") to the MPI adaptive plus local plus global optimization scheme ("MAS"), illustrated in Figure 7.6b.



(a) Energy Delay^2 product



(b) Breakdown of normalized execution time

Figure 7.6: Evaluating the different components of our scheme. Showing (a) normalized (a) Energy Delay^2 product and (b) breakdown of normalized execution time for (i) using the local MPI-driven DVFS only (“MPI_Adapt” or “MA”), (ii) adding local synchronization-aware DVFS module (“MPI_Adapt_LocalSync” or “MALS”), and (iii) adding the global synchronization-aware module (“MPI_Adapt_Sync” or “MAS”).

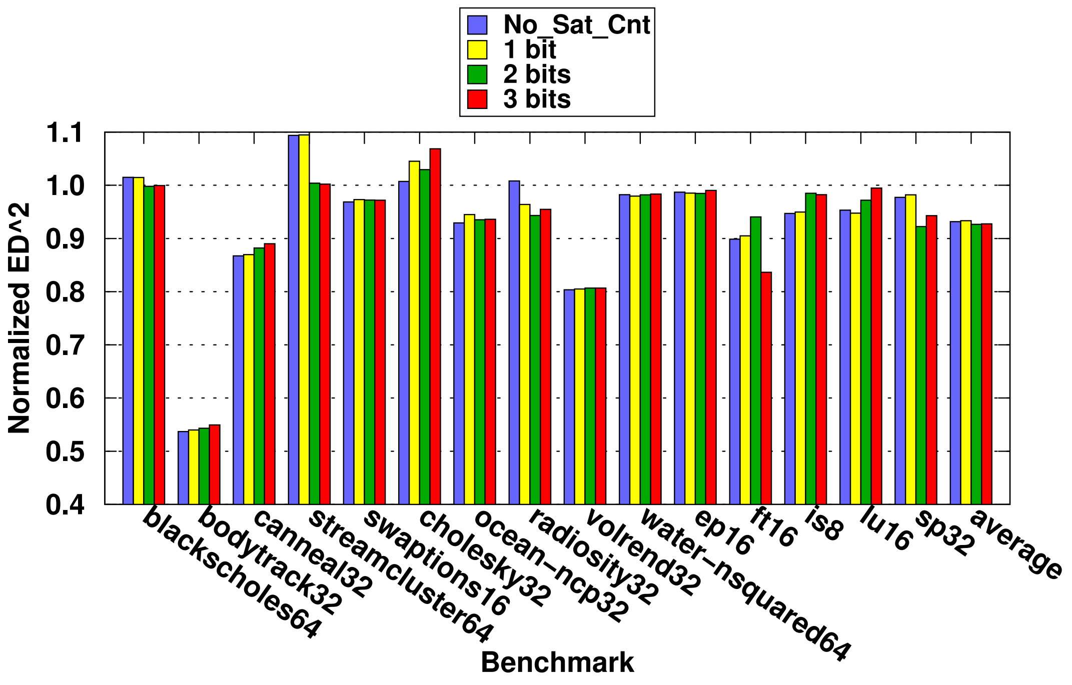


Figure 7.7: Effect of using Saturating Counter Table with different number of bits. Showing normalized Energy Delay^2 product for (a) No saturating counter table (“No_Sat_Cnt”), (b) 1-bit saturating counters (“1 bit”), (c) 2-bit saturating counters (“2 bits”), and (d) 3-bit saturating counters (“3 bits”).

7.2.1.2 Confidence Estimation

Suggestion confidence can mitigate phase mispredictions induced by frequent fluctuations in a thread's performance by requiring a thread's behavior to be consistent over longer time periods before making a prediction. The sensitivity of our scheme under different levels of confidence estimators in the form of varying saturating counter bitwidth is shown in Figure 7.7. Even a 2-bit confidence estimator is able to successfully improve performance by mitigating performance noise in the case of *streamcluster*, *radiosity*, *sp*, and *blackscholes*.

However, while confidence estimation improves prediction performance in the presence of noise it tends to increase the prediction learning time. This is evident in the case of *cholesky*, *is*, *lu*, and to a lesser extent *canneal*, *bodytrack*, and *ocean-ncp*. *ft* shows mixed behavior, with 2-bits performing worse than 1-bit or no confidence estimator but with 3-bits performing significantly better than the others.

It is worth noting that the low latency of the evaluated DVFS mechanism presented in Section 4.3 somewhat lessens the effect of confidence estimation. Under DVFS mechanisms operating at coarser granularities, like the ones present in current state-of-the art multi-processors [59], we expect confidence estimation to be more critical in performance, and hence opt to keep them in our power management system to broaden its applicability.

7.3 Evaluation Under Explicit Thread Only

Finally, we evaluate the applicability of our scheme on multithreaded application with *no* implicit threads. We note at this point that we assume only one explicit thread running in each tile so that we have complete control over each explicit thread. This, however, could easily be alleviated by having voltage and frequency control over each core, and our scheme is not limited in this way. Further, the MPI mappings for implicit threads presented in Table 6.1 are naturally not taken into account in the case of pure explicit thread execution.

The results depicted in Figures 7.8, 7.9, and 7.10, are normalized to the best performing explicit thread *only* version (as opposed to the results shown earlier in this chapter which were normalized to the best performing explicit and implicit thread version). These results clearly show that our scheme remains beneficial even in the absence of implicit threads. This is a testament to the general applicability of our power

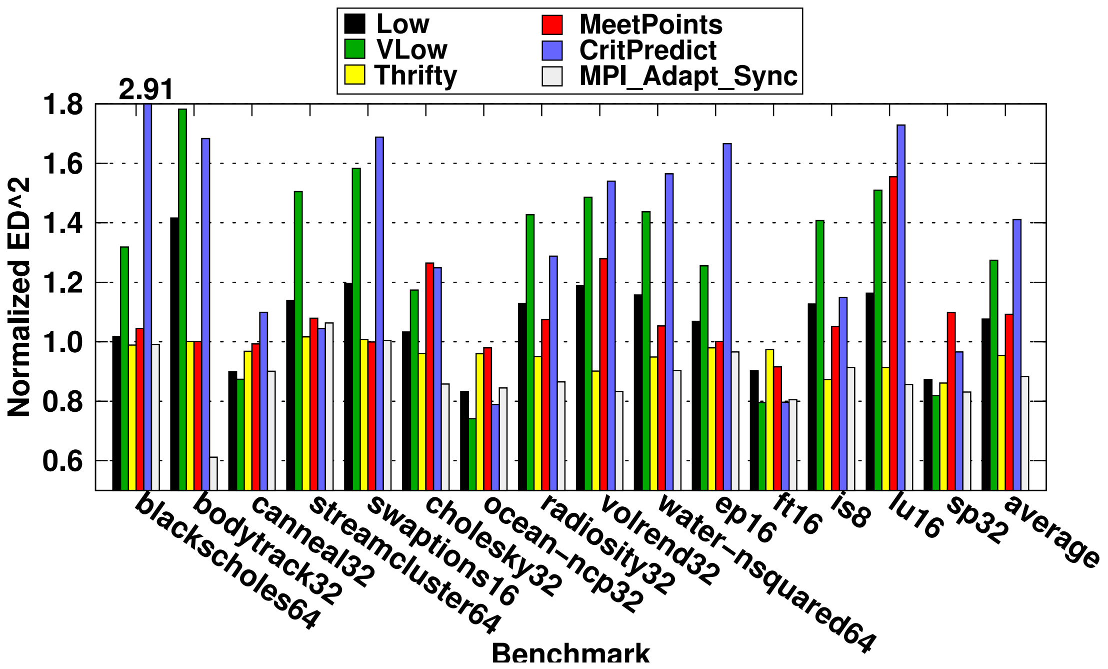


Figure 7.8: Energy Delay^2 product for each power management scheme, normalized to a baseline with no power management. Using explicit threads only.

management scheme. In fact, our scheme is *more* beneficial in the pure explicit thread case showing an average reduction in ED^2 of 12% compared to 8% for the mixed implicit and explicit thread case. This can be attributed to the more energy efficient baseline in the case of having mixed implicit and explicit threads due to reduced synchronization time as explained in Chapter 5. The explicit thread baseline simply has more room for improvement in terms of energy efficiency.

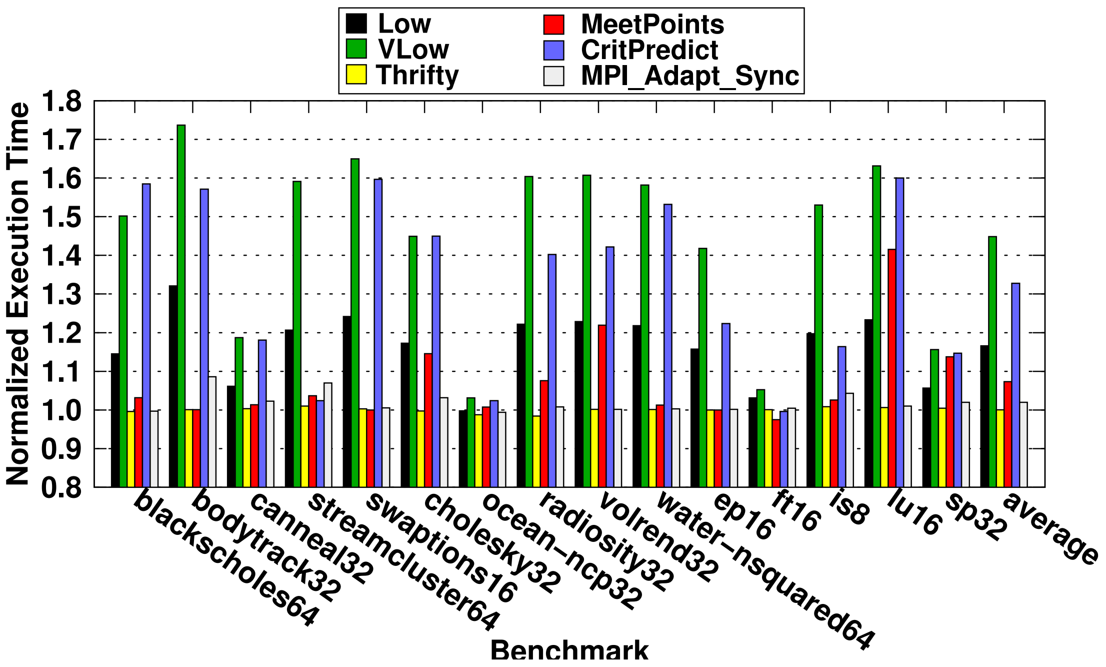


Figure 7.9: Execution time for each power management scheme, normalized to a baseline with no power management. Using explicit threads only.

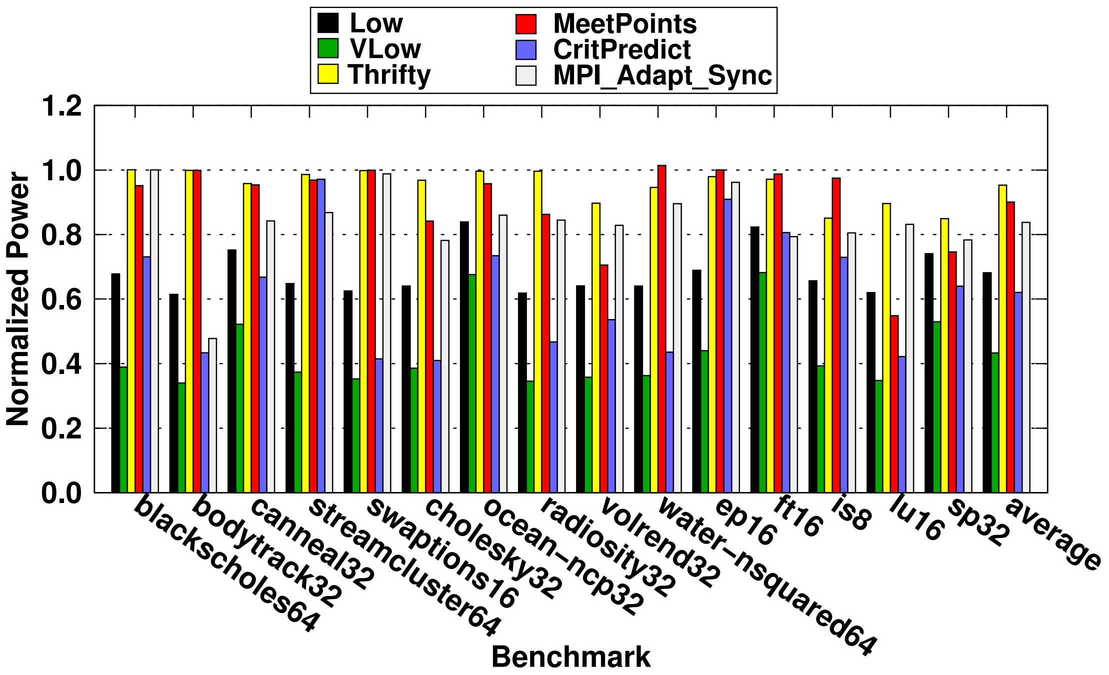


Figure 7.10: Average power consumption for each power management scheme, normalized to the baseline with no power management. Using explicit threads only.

Chapter 8

Related Work

TLS has been a topic of intense investigation over the years, but the vast majority of previous work has applied it to single-threaded applications. The implicit goal of such prior work on TLS, thus, was to achieve scalable parallel performance solely through implicit speculative threads, while our goal is to improve the scalability of explicitly parallel programs. TLS has been applied to individual threads of parallel applications in [112], but its overall impact on the entire application has not been considered; their focus lies in task selection and they only emulate TLS to estimate the overlap potential of each task. The possibility of applying TLS to improve the scalability of parallel applications has been briefly raised in recent studies on coping with Amdahl's law in many-cores (e.g., [46]). However, these have not investigated the architecture implications or the trade-offs involved when combining explicit threads with TLS. To the best of our knowledge, this is the first work to investigate these trade-offs and propose a viable architecture for this purpose.

The closest prior work to ours is [62], where Speculative Parallel-stage Decoupled Software Pipelining (S-PS-DSWP) (originally proposed in [48]) is used to parallelize threads of a parallel application. Speculative Parallel-stage Decoupled Software Pipelining (S-PS-DSWP) [48] is a technique that supports speculative parallel execution of loop iterations, with nested sub-transactions within each iteration. Besides the differences between TLS and S-PS-DSWP as well as between the architectures, the main difference between that work and ours is that we investigate the trade-offs in scalability between employing more outer explicit threads versus more inner speculative implicit threads, and we also propose a mechanism for dynamically and automatically identifying the best trade-off.

Nested explicit and non-speculative parallelism is commonly exploited in the high-performance community through MPI-OpenMP environments. Also, nested explicit speculative parallelism has been proposed with nested transactional memory (e.g., [82]). Unlike our implicit speculative TLS threads, nested transactional memory is not transparent to the user and requires additional programming effort.

Two prior works [85, 86] have considered architectures that support both explicit and TLS threads. They propose the use of explicit parallelism where available, and the use of TLS only outside these explicit parallel regions. Thus they switch between explicit and implicit thread modes, and do not support both types of threads simultaneously in a nested fashion.

In addition to works that consider nested and speculative parallelism, our work is also very much related to current efforts that attempt to mitigate Amdahl's law through other dynamic and transparent means. Core-fusion [55] does so by dynamically "merging" cores together and exploiting ILP when sufficient explicit parallelism is not available. Recent commercial multi-cores incorporate Frequency Boost [50] to shift resources to a subset of cores in order to improve performance when the workload does not provide enough parallelism to utilize all cores. The work in [75] proposed a hardware/software scheme to improve performance of sequential applications using speculative fine-grain multithreading. It uses a clustered architecture, similarly to this work, but only evaluates single threaded applications. The now cancelled ROCK architecture [23] proposed the use of automatic, hardware implicit threads to complement sequential execution by either exposing more ILP or MLP.

Speculative Lock Elision (SLE) is another similar approach [79, 88, 89], that allows explicit threads to run speculatively ahead of a synchronizing operation. SLE tries to overlap some of the work inside critical sections with work being done by other threads outside the critical section, while our approach is to accelerate the execution of the critical section itself. Like our approach, this technique attempts to reduce the amount of time lost in synchronization in the case of large critical sections and load imbalance (Figure 3.5a,c). It offers little benefits, however, for applications limited by regions non proportional to the dataset (Figure 3.5b). Also, unlike our nested approach, SLE requires speculation control across the entire system, which makes the scheme harder to scale to many-core systems.

Finally, [105] speeds up applications with significant critical sections using a static heterogeneous architecture. Our approach, and those mentioned above, provide a more dynamic approach to deal with critical sections and Amdahl's law.

8.1 Power Management

Power management techniques to reduce energy consumption have been extensively studied in prior work. Our focus lies on multithreaded workloads running on a clustered many-core architecture. DVFS schemes for multithreaded applications have primarily targeted workloads using barriers [9, 19, 69, 71].

The thrifty barrier [69] uses the idleness at the barrier to move the faster cores to a low power mode by predicting barrier stall times based on prior behavior. We also apply lower power modes when idling at barriers, but we also employ lower DVFS settings when stalling at other synchronization primitives as well. Moreover, our scheme is applicable to a wide range of parallel workloads and exploits the amenability of program phases to different DVFS operation points throughout a parallel program's execution – both in parallel regions and in synchronization primitives – unlike [69].

Meeting points [19] follows a different approach. It assumes strict Single Program Multiple Data semantics (e.g., traditional OpenMP) and places "meeting points" at the end of each parallel loop. Each thread monitors its current progress based on these meeting points, compares it against the other threads' progress and accordingly throttles down if it detects that it is further ahead in execution. Our scheme is applicable to any parallel workload, however, and is not limited to SPMD programs only.

Thread criticality predictors [9] (TCP) dynamically monitor the cache misses of each thread and give higher priority to the thread that suffers the most misses. The insight behind this is that the thread that is most likely to reach last at the barrier is the slower one, and the one suffering the most misses is an obvious candidate. Unlike TCP, our scheme is not limited to barrier intensive applications alone, but is able to manage a larger set of parallel workloads. Also, we do not rely on predictions to characterize thread criticality; we use current synchronization behavior instead.

The phase-driven "local" component of our power management scheme (see Chapter 6.2.1) is closely related to the work by Icsi et al. [57]. We both characterize phases and drive DVFS management through the memory operations per instruction as an independent metric with respect to frequency. Our scheme aims multithreaded workloads, however, taking into account synchronization on-top of memory behavior.

Chapter 9

Conclusions and Future Work

This chapter presents the conclusions reached by this thesis, and discusses possible future work extensions.

9.1 Summary of Contributions

With the advent of multi-cores, programmers have to endorse parallel programming if they are to exploit the additional hardware resources to improve their applications' performance. Fine-grain parallelism is hard and error-prone, however, and programmers usually avoid parallelizing their applications using fine-grain threading. They instead focus on uncovering parallelism at a coarser granularity which offers a good compromise between performance improvement and development time.

In this thesis we have proposed using implicit speculative parallelism to complement user-level explicit threads. Our scheme is able to improve a parallel application's performance beyond its higher scalability point by using cores that would otherwise be underutilized to run implicit speculative threads. Moreover, given the guaranteed sequential semantics of the TLS protocol, this further parallelization is transparent to the programmers so that they do not have to struggle to further partition and debug the parallel code. Experimental results on a simulated 128-core show that performance improves on top of the highest scalability point by as much as 102%, and 44% on average, for a 4-way cluster and by as much as 85%, and 31% on average, for a 2-way cluster. Significantly, we have further shown that these performance improvements come at virtually no increase in energy consumption. Also, we have presented a comprehensive analysis of performance bottlenecks in the evaluated multithreaded workloads and evaluated their behavior in the presence of different input datasets. Furthermore, we present an auto-tuning mechanism to dynamically choose the number of explicit and implicit threads for *OpenMP* applications which performs within 6% of the static oracle thread allocation.

Finally, we have presented an adaptive, hierarchical power management scheme that is applicable to a wide range of parallel workloads – with and without implicit threads. It comprises two components: (a) a "local" component that follows a thread's memory performance taking into account the difference in behavior between explicit and implicit threads and chooses a locally optimal voltage and frequency pair, and (b) a "global" component that tries to make globally optimal decisions based on the synchronization behavior. Our scheme is able to significantly outperform competing power management schemes on the evaluated platform and workloads and enjoys a significant reduction of Energy Delay^2 product of as much as 46% and 8% on average compared to the baseline. This is due to a reduction in power consumption of as much as 47% and 15% on average with a minimal loss in performance of less than 3% on average. Most importantly, our power management scheme maintains its applicability through all the types of parallel workloads evaluated: barrier-intensive, lock-intensive, and data parallel.

9.2 Future Work

A natural avenue of future research work would be to use other forms of implicit mechanisms to further improve performance of multithreaded workloads. These implicit mechanisms include Helper Threads [22], Runahead execution [83], and Multipath execution [1]. Combining the implicit speculative threads presented in this thesis with Helper Threads, Runahead execution, and Multipath execution could be beneficial for the cases where TLS alone fails to provide benefits. All these mechanisms require similar hardware extensions, and if the support for one is provided, supporting any of the others requires only incremental extensions [115]. The interplay between these implicit mechanisms would also provide interesting research potential in selecting the most beneficial choice of mechanisms to be applied, statically or dynamically. Our prior work in improving sequential application performance by combining Thread-Level Speculation with Helper Threads and Runahead execution [115], and TLS with Multipath execution [114], is a testament to the potential of this future work.

The static task selection heuristic used in our implicit thread proposal in this thesis could be naturally extended and further studied. Prior work in task selection for TLS systems shows significant potential, both in profile driven static approaches and/or dynamic hardware task selection schemes [73]. A good task selection scheme for implicit tasks running on top of explicit threads would require revisiting the cost models of prior approaches (e.g., incorporating synchronization costs). Moreover, sophisticated static or dynamic checkpointing and synchronization schemes [26, 61, 117] could be studied in the context of explicit and implicit threads.

Further, we could extend the power management scheme presented in this thesis to incorporate an elaborate control loop to guarantee it settling in an optimal setting based on an analytical model. Also, our power management scheme could be studied and potentially extended in the context of sequential and multi-programmed workloads to broaden its applicability.

Appendix A

Benchmarks

A.1 PARSEC 2.1

The Parsec benchmarks suite represents modern and emerging parallel workloads. It has been included in our evaluation to study the effect of the schemes presented in this thesis in contemporary and future parallel applications with larger datasets.

A.1.1 blackscholes

The blackscholes application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes Partial Differential Equation (PDE). There is no closed form expression for the Black-Scholes equation and as such it must be computed numerically. The blackscholes benchmark was chosen to represent the wide field of analytic PDE solvers in general and their application in computational finance in particular. The program is limited by the amount of floating-point calculations a processor can perform.

blackscholes stores the portfolio with numOptions derivatives in array OptionData. The program includes file option-Data.txt which provides the initialization and control reference values for 1,000 options which are stored in array data init. The initialization data is replicated if necessary to obtain enough derivatives for the benchmark. The program divides the portfolio into a number of work units equal to the number of threads and processes them concurrently. Each thread iterates through all derivatives in its contingent and calls function BlkSchlsEqEuroNoDiv for each of them to compute its price.

A.1.2 bodytrack

The bodytrack computer vision application is an Intel RMS workload which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence. bodytrack employs an annealed particle filter to track the pose using edges and the foreground silhouette as image features, based on a 10 segment 3D kinematic tree body model. These two image features were chosen because they exhibit a high degree of invariance under a wide range of conditions and because they are easy to extract. An annealed particle filter was employed in order to be able to search high dimensional configuration spaces without having to rely on any assumptions of the tracked body such as the existence of markers or constrained movements. This benchmark was included due to the increasing significance of computer vision algorithms in areas such as video surveillance, character animation and computer interfaces.

The parallel kernels use tickets to distribute the work among threads and balance the load dynamically. The ticketing mechanism is implemented in class TicketDispenser and behaves like a shared counter.

A.1.3 canneal

This kernel was developed by Princeton University. It uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design. SA is a common method to approximate the global optimum in a large search space. Canneal pseudo-randomly picks pairs of elements and tries to swap them. To increase data reuse, the algorithm discards only one element during each iteration which effectively reduces cache capacity misses. The SA method accepts swaps which increase the routing cost with a certain probability to make an escape from local optima possible. This probability continuously decreases during runtime to allow the design to converge. The program was included in the workload selection to represent engineering workloads, for the fine-grained parallelism with its lock-free synchronization techniques and due to its pseudo-random worst-case memory access pattern.

A.1.4 streamcluster

This RMS kernel was developed by Princeton University and solves the online clustering problem: For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. Stream clustering is a common operation where large amounts or continuously produced data has to be organized under real-time conditions, for example network intrusion detection, pattern recognition and data mining. The program spends most of its time evaluating the gain of opening a new center. This operation uses a parallelization scheme which employs static partitioning of data points. The program is memory bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases. Due to its online character the working set size of the algorithm can be chosen independently from the input data. streamcluster was included in the evaluated workloads suite because of the importance of data mining algorithms and the prevalence of problems with streaming characteristics.

A.1.5 swaptions

The swaptions application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a no arbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. Swaptions therefore employs Monte Carlo (MC) simulation to compute the prices. The workload was included in the benchmark suite because of the signicance of PDEs and the wide use of Monte Carlo simulation.

The program stores the portfolio in the swaptions array. Each entry corresponds to one derivative. Swaptions partitions the array into a number of blocks equal to the number of threads and assigns one block to every thread. Each thread iterates through all swaptions in the work unit it was assigned and calls the function HJM Swaption Blocking for every entry in order to compute the price. This function invokes HJM SimPath Forward Blocking to generate a random HJM path for each MC run. Based on the generated path the value of the swaption is computed.

A.2 SPLASH2

The SPLASH2 parallel benchmark suite is the (now ageing) de facto standard in parallel applications. It comprises highly optimized parallel workloads with fine-grain locks and barriers. It has been chosen to allow the evaluation of hand optimized parallel workloads with good scalability.

A.2.1 cholesky

The blocked sparse Cholesky factorization kernel factors a sparse matrix into the product of a lower triangular matrix and its transpose. It is similar in structure and partitioning to the LU factorization kernel (see LU), but has two major differences: (i) it operates on sparse matrices, which have a larger communication to computation ratio for comparable problem sizes, and (ii) it is not globally synchronized between steps.

A.2.2 ocean-ncp

The ocean application studies large-scale ocean movements based on eddy and boundary currents, and is an improved version of the ocean program in the original SPLASH benchmark suite. The major differences are: (i) it partitions the grids into square-like subgrids rather than groups of columns to improve the communication to computation ratio, (ii) grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the nodes that own them, and (iii) it uses a red-black Gauss-Seidel multigrid equation solver, rather than an SOR solver.

A.2.3 radiosity

This application computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method. A scene is initially modeled as a number of large input polygons. Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided into patches as necessary to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step, the patch radiosities are combined via an upward pass through the quadtrees of patches to determine if the overall radiosity has converged. The main data structures represent patches, interactions, interaction lists, the quadtree structures, and a Binary Space Partitioning (BSP) tree which facilitates efficient visibility computation between pairs of polygons. The structure of the computation and the access patterns to data structures are highly irregular. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing. No attempt is made at intelligent data distribution.

A.2.4 volrend

This application renders a three-dimensional volume using a ray casting technique. The volume is represented as a cube of voxels (volume elements), and an octree data structure is used to traverse the volume quickly. The program renders several frames from changing viewpoints, and early ray termination and adaptive pixel sampling are implemented, although adaptive pixel sampling is not used in this study. A ray is shot through each pixel in every frame, but rays do not reflect. Instead, rays are sampled along their linear paths using interpolation to compute a color for the corresponding pixel. The partitioning and task queues are similar to those in raytrace. The main data structures are the voxels, octree and pixels. Data accesses are input-dependent and irregular, and no attempt is made at intelligent data distribution.

A.2.5 water-nsquared

This application is an improved version of the water program in the original SPLASH benchmark suite. This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $O(n^2)$ algorithm (hence the name), and a predictor-corrector method is used to integrate the motion of the water molecules over time. The main difference from the SPLASH program is that the locking strategy in the updates to the accelerations is improved. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end.

A.3 NASPB

The NAS Parallel Benchmarks suite comprises traditional scientific workloads. The *OpenMP* version of NASPB has been chosen to be evaluated in order to study the schemes presented in this thesis under regular *OpenMP* code.

A.3.1 ep

An "embarrassingly parallel" kernel, which evaluates an integral by means of pseudorandom trials. This kernel, in contrast to others in the list, requires virtually no interprocessor communication.

A.3.2 ft

A 3-D partial differential equation solution using FFTs. This kernel performs the essence of many “spectral” codes. It is a rigorous test of long-distance communication performance.

A.3.3 is

An integer sort kernel. This kernel performs a sorting operation that is important in “particle method” codes. It tests both integer computation speed and communication performance.

A.3.4 lu

A regular-sparse, block (5*x*5), lower and upper triangular system solution. This problem represents the computations associated with the implicit operator of a newer class of implicit CFD algorithms, typified at NASA Ames by the code “INS3D-LU”.

A.3.5 sp

Solution of multiple, independent systems of non diagonally dominant, scalar, pentadiagonal equations. SP is representative of computations associated with the implicit operators of Computational Fluid Dynamics (CFD) codes such as “ARC3D” at NASA Ames.

Bibliography

[1] Pritpal S. Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Multipath execution: Opportunities and limits. In *Intl. Conf. on Supercomputing (ICS)*, pages 101–108, July 1998.

[2] David H. Albonesi, Rajeev Balasubramonian, Steven G. Dropsho, Sandhya Dwarkadas, Eby G. Friedman, Michael C. Huang, Volkan Kursun, Grigorios Magklis, Michael L. Scott, Greg Semeraro, Pradip Bose, Alper Buyuktosunoglu, Peter W. Cook, and Stanley E. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36:49–58, December 2003.

[3] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Addison-Wesley, 1994.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conf.*, pages 483–485, April 1967.

[5] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through epi throttling. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[6] Murali Annavaram, Ryan Rakvic, Marzia Polito, Jean-Yves Bouguet, Richard Hankins, and Bob Davies. The fuzzy correlation between code and performance predictability. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 93–104, December 2004.

[7] David H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod Fatoohi, Paul O. Frederickson, T. A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira Weeratunga. The

nas parallel benchmarks – summary and preliminary results. In *Conference on Supercomputing (SC)*, pages 158–165, December 1991.

[8] Major Bhadauria, Vincent M. Weaver, and Sally A. McKee. Understanding parsec performance on contemporary cmps. In *Intl. Symp. on Workload Characterization (IISWC)*, pages 98 –107, October 2009.

[9] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[10] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *Intl Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 99–108, June 2002.

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, October 2008.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356 –368, November 1994.

[13] Colin Blundell, E Christopher Lewis, and Milo M K Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Wksp. on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[14] Shekhar Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference (DAC)*, pages 746–749, June 2007.

[15] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conference (DAC)*, pages 338–342, 2003.

[16] David Brooks and Margaret Martonosi. Dynamic thermal management for high-performance microprocessors. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, volume January, pages 171 –182, 2001.

[17] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[18] Mark Bull. *Personal Communication*. Edinburgh Parallel Computing Center, May 2011.

[19] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 240–249, October 2008.

[20] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 227–238, June 2006.

[21] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 97–108, February 2007.

[22] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (ssmt). In *Intl. Symp. on Computer Architecture (ISCA)*, pages 186–195, June 1999.

[23] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Hkan Zeffer, and Marc Tremblay. Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's ROCK processor. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 484–495, June 2009.

[24] Tiberiu Chelcea and Steven M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Design Automation Conference (DAC)*, pages 21 – 26, June 2001.

[25] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 13–24, June 2000.

[26] Christopher B. Colohan, Anastassia Ailamaki, J. Gregory Steffan, and Todd C. Mowry. Tolerating dependences between large speculative threads via subthreads. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 216–226, June 2006.

[27] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Intl. Conf. on Supercomputing (ICS)*, pages 157–166, June 2006.

[28] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 250–259, October 2008.

[29] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5:46–55, January 1998.

[30] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42:36 –42, December 2009.

[31] Bronis R. de Supinski. *Personal Communication*. Lawrence Livermore National Laboratory, May 2011.

[32] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Intl. Symp. on Low Power Electronics and Design (ISLPED)*, pages 207–212, August 2007.

[33] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 233–244, June 2002.

[34] Saurabh Dighe, Sriram R. Vangal, Paolo A. Aseron, S. Kumar, Tiju Jacob, Keith A. Bowman, Jason Howard, James Tschanz, Vasantha Erraguntla, Nitin Borkar, Vivek De, and Shekhar Borkar. Within-die variation-aware dynamic-voltage-frequency scaling core mapping and thread hopping for an 80-core processor. In *International Solid-State Circuits Conference (ISSCC)*, pages 174 –175, February 2010.

[35] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 78–88, 2006.

[36] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–231, September 2003.

[37] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Intl. Symp. on Computer architecture (ISCA)*, pages 362–370, June 2010.

[38] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computer (TC)*, 45:552–571, May 1996.

[39] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 164–173, June 2005.

[40] Stanley L. C. Fung and J. Gregory Steffan. Improving cache locality for thread-level speculation. In *Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 32–32, April 2006.

[41] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 191–202, February 2003.

[42] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 195–205, February 1998.

[43] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31:532–533, May 1988.

[44] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The stanford hydra cmp. *IEEE Micro*, 20:71–84, March 2000.

[45] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, October 1998.

[46] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33 –38, July 2008.

[47] Jason Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Intl. Solid-State Circuits Conference (ISSCC)*, February 2010.

[48] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *Intl. Symp. on Code generation and optimization (CGO)*, pages 121–130, April 2010.

[49] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: Application to energy reduction. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 157 – 168, June 2003.

[50] Intel. *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors*, White Paper, November, 2008.

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

[52] Nikolas Ioannou and Macelo Cintra. Complementing explicit coarse-grain parallelism with implicit speculative parallelism. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 284–295, December 2011.

[53] Nikolas Ioannou, Michael Kauschke, Matthias Gries, and Macelo Cintra. Phase-based application-driven power management on the single-chip cloud computer. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 131 –142, October 2011.

[54] Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, Paris Yiapanis, Adam Pocock, Gavin Brown, Mikel Lujan, Ian Watson, and Marcelo Cintra. Toward a more accurate understanding of the limits of the tls execution

paradigm. In *Intl. Symp. on Workload Characterization (IISWC)*, pages 1 –12, December 2010.

[55] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martínez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Intl. Symp. on Computer architecture (ISCA)*, pages 186–197, June 2007.

[56] Canturk Isci, Alper Buyuktosunoglu, C.-Y. Cher, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 347–358, December 2006.

[57] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 359 –370, December 2006.

[58] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–70, June 2004.

[59] Opher Kahn and Bob Valentine. *Intel Next Generation Microarchitecture Code-name Sandy Bridge: New Processor Innovations*, Intel Developer Forum, 2010.

[60] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Intl. Conf. on Supercomputing (ICS)*, pages 24–35, June 2006.

[61] Salman Khan, Nikolas Ioannou, Polychronis Xekalakis, and Marcelo Cintra. Increasing the energy efficiency of tls systems using intermediate checkpointing. In *Intl. Conf on High Performance Computing (HiPC)*, pages 1–11, December 2011.

[62] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 3–14, December 2010.

[63] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 123 –134, February 2008.

[64] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Intl. Conf. on Supercomputing (ICS)*, pages 85–92, July 1998.

[65] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multicore architectures: Understanding mechanisms, overheads and scaling. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 408–419, June 2005.

[66] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance evaluation of the omni openmp compiler. In *Intl. Symp. on High-Performance Computing(ISHPC)*, pages 403–414, October 2000.

[67] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33 – 42, May 2006.

[68] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 148–159, May 1990.

[69] Jian Li, Jose F. Martinez, and Michael C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 14–23, February 2004.

[70] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *Supercomputing Conf. (SC)*, December 2006.

[71] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir, and Mary Jane Irwin. Exploiting barriers to optimize power consumption of cmps. In *Intl. Parallel and Distributed Processing Symp.(IPDPS)*, pages 5.1–, April 2005.

[72] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure.

In *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 158–167, March 2006.

[73] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 462–473, June.

[74] Peter Macken, Marc Degrauwe, Mark Van Paemel, and Henry Oguey. A voltage reduction technique for digital systems. In *Intl. Solid-State Circuits Conf. (ISSCC)*, pages 238–239, February 1990.

[75] Carlos Madriles, Pedro López, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raúl Martinez, and Antonio Gonzalez. Boosting single-thread performance in multi-core systems through fine-grain multithreading. In *Intl. Symp. on Computer architecture (ISCA)*, pages 474–483, June 2009.

[76] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A fine-grain thread decomposition scheme for speculative multithreading. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–25, September 2009.

[77] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 14–27, June 2003.

[78] Pedro Marcuello and Antonio González. Clustered speculative multithreaded processors. In *Intl Conf. on Supercomputing (ICS)*, pages 77–84, June 1999.

[79] Jose Martinez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, October 2002.

[80] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9:21–65, February 1991.

[81] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Intl. Symp. on Computer architecture (ISCA)*, pages 69–80, June 2007.

[82] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 359–370, October 2006.

[83] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 129–140, February 2003.

[84] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[85] Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Intl. Conf on Supercomputing (ICS)*, pages 368–380, June 2001.

[86] Leo Porter, Bumyong Choi, and Dean M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 313–324, September 2009.

[87] Carlos García Quinones, 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 *Conf. on Programming Language Design and Implementation (PLDI)*, pages 269–279, June 2005.

[88] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Intl Symp. on Microarchitecture (MICRO)*, pages 294–305, December 2001.

[89] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, October 2002.

[90] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Intl. Symp. on Code Generation and Optimization (CGO)*, pages 175–184, April 2008.

[91] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 302–313, June 2009.

[92] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

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

[94] 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 *Intl. Conf. on Supercomputing (ICS)*, pages 179–188, June 2005.

[95] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Intl. Conf. on Supercomputing (ICS)*, pages 460–469, June 2009.

[96] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *Conf. on LISP and Functional Programming (LFP)*, pages 202–211, August 1986.

[97] Greg Semeraro, Grigorios Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 29–40, February 2002.

[98] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 165–176, October 2004.

[99] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 336–349, June 2003.

[100] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 414–425, June 1995.

[101] Robert Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in mpi programs on an energy-constrained, power-scalable cluster. In *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[102] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 1–12, June 2000.

[103] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23:253–300, August 2005.

[104] J.G. Steffan and T.C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 2–13, January 1998.

[105] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. An asymmetric architecture for accelerating critical sections. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, March 2009.

[106] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[107] David Tarjan, Shyamkumar Thoziyoor, and Norman P. Jouppi. Cacti 4.0. Technical report, Compaq Western Research Lab., 2006.

[108] Josep Torrellas, Monica S. Lam, and John L. Hennessy. *IEEE Transactions on Computer (TC)*, 43(6):651 –663, June 1994.

[109] Marc Tremblay and Shailender Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc processor. In *International Solid-State Circuits Conference (ISSCC)*, pages 82 –83, February 2008.

[110] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. In *International Solid-State Circuits Conference (ISSCC)*, pages 98–589, February 2007.

[111] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 81–92, December 1998.

[112] Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 79–89, March 2007.

[113] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Intl. Symp. on Computer Architecture (ISCA)*, pages 24 – 36, June 1995.

[114] Polychronis Xekalakis and Marcelo Cintra. Handling branches in tls systems with multi-path execution. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 1–12, February 2010.

[115] Polychronis Xekalakis, Nikolas Ioannou, and Marcelo Cintra. Combining thread level speculation, helper threads and runahead execution. In *Intl. Conf. on Supercomputing (ICS)*, pages 410–420, June 2009.

[116] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 261–272, February 2007.

[117] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 171–183, October 2002.