A Hardware/Software Approach for Alleviating Scalability Bottlenecks in Transactional Memory Applications

by

Geoffrey Wyman Blake

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Professor Trevor N. Mudge, Chair
Professor Dennis M. Sylvester
Associate Professor Scott Mahlke
Assistant Professor Thomas F. Wenisch

To my wife Jennifer

ACKNOWLEDGEMENTS

There are almost too many people to thank that have helped me survive my trials through the PhD program here at the University of Michigan.

I would first like to thank my advisor Professor Trevor Mudge who took a chance taking me on as a student in the Fall of 2005 and providing funding when I was still unsure if I could survive in the program. His hands off advising style was a perfect fit that allowed me to pursue what I found interesting, but still hands on enough to help me ask the right questions about what I was doing to continue making progress toward eventually graduating.

Thanks to Taeho Kgil, a senior graduate student I used as my surrogate advisor when Trevor was on sabbatical my first year. His patience in answering my endless stream of questions about all things regarding research and the state-of-the-art in computer architecture was invaluable in pointing me towards finding my niche.

I would like to thank Ronald Dreslinski for spending numerous hours editing papers with me.

To all my lab-mates over the years: Jeff Ringenberg, Mark Woh, Yuan Lin, Gabe Black, Ali Said, Lisa Hsu, Kevin Lim, Korey Sewell, Tony Gutierrez and Joe Pusdesris. Thanks for the many conversations, whether it be research related or just anything at all, to pass the time when working seemed too dull that day.

Finally, I could not have succeeded without the support of my family and friends. Thanks to my parents for supporting my decision to pursue this degree. Thanks to Steve Plaza for first planting the idea I should get a PhD instead of just taking the easy way and settling for a Masters. To Chris McLean for moral support on numerous life issues and the many others who I've befriended along the way. Lastly, to my wife Jennifer who I could not have completed this degree without. She helped me balance life and research when I was too focused on getting just one more experiment done after hours and hours of not getting anywhere. Jennifer also kept me motivated to complete all the work required to finish, for that I cannot thank her enough.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF EXAMPLES

Example

ABSTRACT

A Hardware/Software Approach for Alleviating Scalability Bottlenecks in Transactional Memory Applications

by

Geoffrey Wyman Blake

Chair: Trevor N. Mudge

Scaling processor performance with future technology nodes is essential to enable future applications for devices ranging from smart-phones to servers. But the traditional methods of achieving that performance through frequency scaling and single-core architectural enhancements are no longer viable due to fundamental scaling limits. To continue scaling performance, parallel computers in the form of Chip Multi-processors (CMPs) are now prevalent. The evolution from single-threaded processors to CMPs has moved the use of parallel programming from niche problem domains to general problems.

Producing scalable, efficient parallel programs is challenging. One challenging area is scalable synchronization to shared data structures. Traditional synchronization methods synchronize multiple threads accesses to shared data structures by restricting memory accesses to a defined correct set of interleavings which the programmer must enforce by hand. It can take many years for even expert programmers to use traditional synchronization methods to craft an efficient, scalable and correct scheme to synchronize access to data structures in a complex program. The use of CMPs in almost all forms of computers has led researchers to look into methods to make scalable synchronization more tractable, increasing programmer productivity by raising the level of abstraction. One proposal to raise the level of abstraction is to use "Transactional Programming", to represent atomic sections of code. Transactions abstract synchronization to shared data structures as large, coarse-grained sections of code that execute atomically with respect to one another or not all. This relieves the programmer of identifying all appropriate memory interleavings. Transactional programming can be supported underneath in many ways, one of which is to use a

"Transactional Memory" (TM) system that can efficiently support transactions.

One main problem with TM systems is scalability bottlenecks. Performance scaling stops or reverses when moving from small to large CMPs when transactional applications are written to emulate future average programmer practices—namely using large transactions instead of crafting transactions to behave like traditional synchronization methods. This happens because transactions as represented in the TM system may be dependent on each other—accessing the same data and therefore must serialize—without the programmer being knowledgable about these dependencies due to the abstraction hiding system details.

This thesis develops a hardware/software approach to alleviate scalability bottlenecks in large TM enabled CMPs, while maintaining the level of abstraction—large, coarse-grained sections of code—presented in transactional programming. I first introduce *"Proactive Transaction Scheduling"* (PTS), a technique that dynamically profiles the parallel code as it executes to determine an order transactions should execute in to maintain forward progress and increase the parallel utilization by hiding dependencies among transactions. To further increase scaling I then propose using PTS to automatically determine which transactions must be accelerated by increasing the frequency of the core the transaction is executing on for short periods of time. This allows a transaction that is causing other dependent transactions to wait to complete faster and promote scaling. I then show PTS can be used to determine how to partition resources in a Multi-threaded processor core to get better scaling in the overall system over a fair partitioning of resources.

CHAPTER 1

Introduction

In the early 2000s chip manufacturers began to migrate from single-threaded processors. Chip architects were struggling to improve single-threaded performance without increasing the complexity of the design or the power consumption needed to maintain high clock frequencies. This was very apparent in Intel's Pentium 4 [58] processor. It was an extremely complicated single-threaded design with more than 30 pipeline stages that implemented a vast array of techniques to enhance performance. The Pentium 4 design was also extremely power hungry. Transistors, however, were still scaling to smaller geometries allowing for double the amount of transistors approximately every two years. Researchers began to experiment with Chip Multi-processors (CMPs) as an alternative method to scale performance. Examples such as the Stanford Hydra [63] and Compaq Piranha [25] were proposed as a way to make profitable use of the increasing number of transistors available each process generation. CMPs increase available raw compute performance by placing multiple processor cores on die. Because CMPs rely on explicit parallelism in the program code for performance, overall design complexity can be reduced by reusing core designs as well as reducing power consumption by not increasing frequency. The initial research in CMPs was followed closely by IBM releasing the first commercial CMP: the POWER4 [1]. Intel and AMD followed closely thereafter with CMPs of their own [3, 2].

CMPs are now prevalent across many domains from mobile chips for smart phones to servers. AMD and Intel have at the time of this writing, server chips with 8 to 16 processors [11, 12]. ARM partners are releasing chips for mobile devices such as tablets and smart phones with 2 to 4 processors [10]. Processors for more specialized applications such as network processing chips have even more cores per die. Chips from Cavium have up to 32 processors [13], while chips from Tilera have greater than 100 cores [5].

The influx of CMPs has led to a resurgence of research into parallel programming methodology. As Sutter [104] has written, the free lunch is over for transparent performance increases from processor makers due to frequency and architecture enhancements.

New software programming techniques and architectures are needed to leverage the compute performance in CMPs that is still largely untapped as shown in [29] for applications outside the normal domain for parallel programs. Before CMPs, parallel processing was mainly used in domains that were easily parallelized such as scientific computing and commercial server applications. Because of this small domain, methodologies for programming these machines remained low-level and accessible only to a small group of experts. Now that CMPs are being used in many more computing platforms all the way down to smartphones, low-level programming methodologies are insufficient as more programmers will now be programming CMPs. Making parallel programming tractable to more developers is of main concern among researchers in academia and industry.

One of the areas of concern is performing scalable, yet easy and safe synchronization for accessing shared data. Currently, synchronization in parallel programming is still very low-level and only tractable to a select group of experts. Programmers are required to synchronize separate threads accesses to shared data by using primitives such as locks, semaphores, monitors, and condition variables that restrict access to memory locations in programmer defined orders. Use of these primitives is difficult as they are opaque datatypes with meaning only to the programmer. Composing the correct synchronization primitives to protect critical sections is difficult, especially if the critical section accesses multiple pieces of shared data. This can be difficult as specific acquire and release orders must be maintained to prevent deadlock conditions from occurring. Therefore these primitives must be carefully documented to prevent future synchronization errors. Other problems with traditional synchronization include priority inversion where a low priority thread blocks a high priority thread from executing because it is holding a lock. Race conditions are an additional problem caused by omitting use of the proper locks leading to difficult to find bugs that occur occasionally. As an alternative to these primitives, researchers have also explored using lockless data-structures [66] that use low-level ISA provided atomic instructions. Lockless data-structures can eliminate priority inversion. In many cases lockless algorithms are more complicated and even slower than using the above mentioned synchronization primitives.

Using synchronization primitives is very hard to do effectively and in a scalable manner. Synchronization primitives must be used in a fine grained fashion, i.e. multiple locks used to protect shared accesses, to have scalable synchronized code. If one lock is used to synchronize a program, then it will not be able to scale to many processors. Because CMPs will continue to get bigger as more transistors become available, performance will become a problem with using traditional synchronization for most programmers. Developing scalable, fast algorithms using these primitives is very difficult. For example, a paper

by Ellis [53] shows a scalable and fast locking algorithm for inserting and deleting from an AVL tree. The algorithm itself is complicated using three classes of dynamically created locks and requiring very specific orders of acquires and releases to enable a scalable deadlock free algorithm. If one of the locks is taken in the wrong order, or omitted, deadlock or hard-to-track race conditions can occur that are difficult to debug. Because the current methods of providing synchronization are complicated, low-level and time consuming to implement, only particular applications that absolutely require parallel operation have been profitably multi-threaded using current synchronization primitives. For example, it has taken many years of work to transition the locking algorithms in the Linux v2.4 kernel to support scalable fine grained locking for future CMPs in the Linux v2.6 kernel as described by Hoffman et al. [69]. These applications include: scientific applications, web-servers, databases and operating systems.

Low-level primitives are great for expert programmers. Experts can precisely control the behavior of their parallel programs and extract the maximum performance. Conversely, it is not a viable option for the remainder of the programming community. This has led researchers in industry and academia to look at raising the level of abstraction to make synchronization in parallel programs more accessible to programmers other than a few experts. One promising technique that researchers are actively investigating to raise the level of abstraction for synchronization is enabling Transactional Programming.

1.1 Transactional Programming

Transactional programming takes ideas from the database community and applies it to traditional program synchronization. The initial idea for transactional programming was proposed first by Lomet [81]. Transactional programming provides the following desirable semantics for synchronization that raise the level of abstraction for programmers: 1) Critical sections represented by transactions are atomic, they complete in entirety or not at all 2) Transactions are isolated and outside transactions cannot see interim updates, 3) Transactions are placed around code instead of using opaque data types such as locks to guard data, 4) Provides coarse-grained semantics. Transactional programming theoretically allows the programmer to concentrate more on defining where the critical sections should be in a coarse grained manner, but not have to worry about scalability, atomicity, and isolation.

Transactional programming can be seen in the same light for parallel programming as managed languages that use garbage collection for memory management are for sequential programming [60]. An example of the easier semantics is presented in Example 1.1. Example 1.1 assumes that a programmer wants to insert multiple elements into a graph data

structure represented as a 2-D matrix and have the inserts all happen atomically at once while being scalable. As seen in the example pseudo-code on the left of the example, with transactional programming all the programmer is required to do is encapsulate the operation he wants to be atomic and the underlying implementation that provides the transactions takes care of ensuring atomicity, isolation and scalability. On the other hand, the locking version of the code on the right in Example 1.1 needs to use multiple locks for fine grained synchronization. If a single lock was used, no concurrent updates could happen. With fine grained locking, there is also the very real chance of errors. Example 1.1 illustrates this by having thread 1 and thread 2 take locks in an incorrect order. By taking the locks on locations $[5,5]$ and $[4,5]$ in different orders for the two threads, deadlock is possible. A study by Rossbach et al. [93] tested whether the benefits between transactional programming and traditional synchronization existed for a group of non-expert programmers. The study showed that transactional programming was less prone to errors than constructing fine-grained synchronization algorithms. The verdict was inconclusive amongst the tested programmers if transactional programming was any easier over traditional synchronization. Still, this is a promising result, it shows that transactional programming is a useful abstraction. It shows that raising the level of abstraction does allow novice programmers easier access to producing correct synchronization in parallel programs while maintaining acceptable performance.

1.2 Transactional Memory

For transactional programming to become acceptable it has to be both easy to program and high performance. High performance means that transactional programming should perform near or better than locks in applications. To support transactional programming, Transactional Memory (TM) has been proposed to provide high-performance transactions to the programmer.

Transactional Memory was first proposed by Herihly and Moss [67] as an architectural extension for writing lockless data structures easier, and not an interface to transactional programming originally. Research that followed by numerous authors began proposing TM as a potential method to enable transactional programming. TM systems are used to provide the properties necessary for fast and efficient transactional programming: 1) Critical sections are atomic, they complete in entirety or provide the appearance nothing happened and retry and 2) Critical sections are isolated and outside threads cannot see interim updates that may cause hard to track bugs, and 3) Enables fine grained synchronization performance by tracking individual memory accesses. These guarantees provide sufficient support for

Example 1.1 Example pseudo-code illustrating the semantic usage of Transactional Programming compared to fine grained synchronization using locks for updating a graph data-structure represented as a 2-D matrix.

```
1  //global variables
2  int graph[100][100];
3
4  void* threadwork()
5  {
6    ...
7    TM_BEGIN;
8    insertIntoGraph(5,5,25);
9    insertIntoGraph(4,5,50);
10   insertIntoGraph(99,99,1);
11   TM_END;
12   ...
13 }
14
15 void* threadwork2()
16 {
17   ...
18   TM_BEGIN;
19   insertIntoGraph(4,5,15);
20   insertIntoGraph(5,5,50);
21   insertIntoGraph(98,98,2);
22   TM_END;
23   ...
24 }
25
26 void insertIntoGraph(idx1,
27                      idx2,
28                      val)
29 {
30   graph[idx1][idx2] = val;
31 }
```

```
1  //global variables
2  int graph[100][100];
3  mutex global_locks[100][100];
4
5  void* threadwork()
6  {
7    ...
8    lock(global_locks[5][5]);
9    lock(global_locks[4][5]);
10   lock(global_locks[99][99]);
11   insertIntoGraph(5,5,25);
12   insertIntoGraph(4,5,50);
13   insertIntoGraph(99,99,1);
14   unlock(global_locks[5][5]);
15   unlock(global_locks[4][5]);
16   unlock(global_locks[99][99]);
17   ...
18 }
19
20 void* threadwork2()
21 {
22   ...
23   lock(global_locks[4][5]);
24   lock(global_locks[5][5]);
25   lock(global_locks[99][99]);
26   insertIntoGraph(4,5,15);
27   insertIntoGraph(5,5,50);
28   insertIntoGraph(98,98,2);
29   unlock(global_locks[4][5]);
30   unlock(global_locks[5][5]);
31   unlock(global_locks[99][99]);
32   ...
33 }
34
35 void insertIntoGraph(idx1,
36                      idx2,
37                      val)
38 {
39   graph[idx1][idx2] = val;
40 }
```

implementing a transactional programming infrastructure. TM prototypes have been by implemented by many researchers in a number of different methods. TM can be implemented a software runtime layer that is used by the programmer or the compiler for Software Transactional Memory (STM), as architectural extensions to the processor and memory systems as Hardware Transactional Memory (HTM), or a mix of limited hardware supported with a software runtime layer to create a Hybrid Transactional Memory (HyTM).

STM systems can run on unmodified hardware and use compilers to generate the proper traditional synchronization primitives to provide the transactional programming interface. Unfortunately, they are extremely slow as shown by Cascaval et al. [40]. This has limited the impact of STM as it is much slower than traditional primitives even though it offers the easier semantics of transactional programming. HyTM and HTM systems on the other hand natively support transactional programming primitives (the transaction). This allows them to execute transactions fast and efficiently, making transactional programming competitive with locks. HTM systems in particular though can still suffer from conditions where their performance is much worse than locks. This problem is what this thesis aims to solve. These conditions are presented in the next section. For the remainder of this thesis, unless otherwise state, TM will refer to a system that supports transactional programming on an HTM system.

1.3 Scalability Bottlenecks

A product of raising the level of abstraction for synchronization operations in parallel programming is the increased chance for scalability bottlenecks. Scalability bottlenecks in the context of this thesis are critical sections that hold up other critical sections for long periods of time. This slows down the system by not making profitable use of parallel resources. Figure 1.1 shows how scalability bottlenecks can affect a parallel program's execution by causing long periods of idle time. With the number of cores likely to keep increasing on die, scalability bottlenecks will prevent these extra cores from being used profitably and solutions must be found to either eliminate them or hide the effects of these bottlenecks.

In TM, especially Hardware Transactional Memory (HTM), scalability bottlenecks must be minimized as it can suffer less than serial performance if contention is high. The main argument for HTM is that it has higher performance than locks as well as natively supporting the conceptually easier to programming technique, transactional programming. Contention is when two or more transactions try to modify the same memory locations, also know as a conflict, and cause one or more transactions to abort and try executing

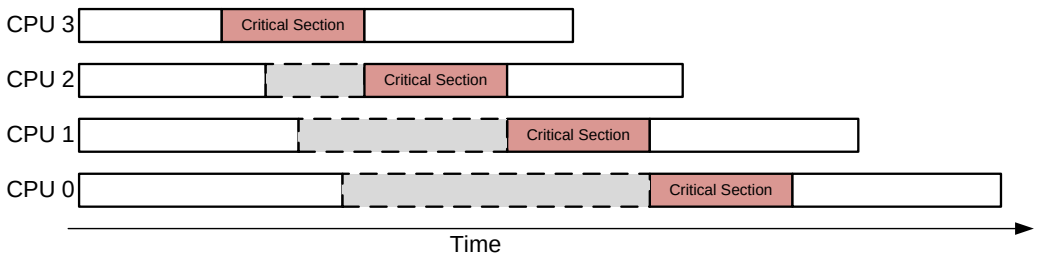


Figure 1.1: Illustration of the serializing effects of scalability bottlenecks.

	Delaunay	Genome	Kmeans	Vacation
2	1.3	1.7	1.9	1.9
4	2.0	2.9	3.8	3.9
8	3.0	1.8	6.6	5.7
16	3.5	0.4	6.7	7.0
	Intruder	SSCA2	Labyrinth	Yada
2	1.5	1.6	1.7	1.3
4	1.5	2.8	3.0	1.9
8	0.8	4.2	3.8	2.8
16	0.3	5.7	6.0	3.7

Table 1.1: Speedup observed for STAMP Benchmarks with simple Randomized Backoff contention manager for a 16 processor system using a LogTM like Transactional Memory system.

the transaction again. This can happen for applications that are written with large coarse grained transactions, such as the STAMP benchmarks, when using simplistic contention management. Table 1.2 shows for the STAMP benchmarks that contention can be very high for its applications for a large 16 core CMP (in Table 1.2 contention is measured as number of transactions aborted divided by the number of transactions began). This in turn leads to very poor performance as seen in Table 1.1. This is especially evident in the *Kmeans*, *Genome* and *Intruder* benchmarks from the STAMP benchmark suite. As the number of processors increase, the contention can increase to a point where performance either plateaus or becomes worse than serial at 16 processors. Under low contention, many of the STAMP applications scale well. Because TM supports the transactional programming abstraction for synchronization, it should not be expected of programmers to have to tune the transactions to the TM implementation in their program to eliminate contention. It may also be the case that they cannot tune transactions as they may be contained in a library that is a black box. Researchers must find ways to minimize the effects of scalability bottlenecks that may be present in transactional code.

	Delaunay	Genome	Kmeans	Vacation
2	29.8%	0.5%	<0.1%	0.1%
4	45.5%	2.6%	0.2%	0.6%
8	57.7%	23.3%	3.8%	4.0%
16	73.4%	49.7%	20.6%	10.4%
	Intruder	SSCA2	Labyrinth	Yada
2	2.5%	<0.1%	7.6%	21.6%
4	12.7%	<0.1%	6.2%	34.1%
8	38.2%	<0.1%	10.5%	45.3%
16	70.1%	<0.1%	18.2%	54.6%

Table 1.2: Contention observed for STAMP Benchmarks with simple Randomized Backoff contention manager for a 16 processor system using a LogTM like Transactional Memory.

1.4 Contributions

The opposing goals of architects and software writers makes developing for transparent solutions for alleviating scalability bottlenecks very important. Raising the level of abstraction is required to make synchronization operations in parallel programming easier and tractable. This means that programmers will not be able to (or expected to) optimize their programs to the degree expert parallel programmers can achieve with low-level primitives. This thesis contributes an approach using a combination of minimal hardware and software runtime that transparently alleviates scalability bottlenecks and lets more cores be profitably used for an assumed LogTM [84] like Hardware Transactional Memory system that supports transactional programming.

1.4.1 Proactive Transactional Memory Scheduling

I first propose a technique called Proactive Transaction Scheduling (PTS). Many contention managers for transactional memories assume that transaction conflicts are a rare and random event. Therefore they employ reactive contention managers built on simple randomized backoff schemes. On a conflict, reactive contention managers will simply abort one of the transactions and give a random time to wait before restarting. Reactive managers do work well for benchmarks that have a low amount of contention as they incur low overhead. As seen in Table 1.2, low contention is not always the case. In fact, high contention is sometimes common, and should be assumed if TM becomes widely used. Novice programmers will not always write applications with inherently low amounts of contention. Bottlenecks will become more likely for higher processor counts—as seen in

the benchmark suite tested.

PTS is a contention management strategy for HTM systems that approaches the problem of high contention among coarse grained transactions as a scheduling problem. Instead of reacting to conflicts as they happen by assuming they are rare, PTS instead assumes that conflicts are common. PTS tracks the conflicts seen by each transaction and uses the resulting dependency graph to make predictions about the future conflicts each transaction may see if executed. It uses this conflict information to schedule transactions to run only when it is predicted the currently running transactions will not conflict with the transaction waiting to execute. To keep utilization of the processor cores as high as possible, when a transaction is predicted to conflict with another in the system, a new transaction is swapped in by calling the Operating System to switch in a new thread to execute. This means the system is overcommitted (more threads than cores). This is a realistic assumption as many applications use more threads than cores, because threads are expected to occasionally block and the extra threads can use the idle cores that would otherwise be wasted. I show in this thesis that PTS is a viable contention management scheme, and greatly reduces contention by hiding dependencies among transactions, increasing utilization of cores and alleviating scalability bottlenecks. PTS is capable of doing so transparently to the programmer, and not requiring transactions to be tuned.

1.4.2 Identifying Proble Critical Sections for Boosting

A key observation for CMPs is that symmetrical systems are ill-suited for dealing with applications that show-case a significant number of scalability bottlenecks. Instead an asymmetrical system is shown to be better suited for executing programs with scalability bottlenecks. An asymmetrical system has a mix of fast and slow cores. To reduce scalability bottlenecks a critical section that has many dependencies could execute on the faster core to complete quicker and reduce the amount time other critical sections may spend waiting on it. The main challenges for this type of system are the following: 1) Determining the critical section causing bottlenecks needing acceleration, 2) How to build the asymmetrical system, 3) How to place the critical section onto the core that will provide the acceleration.

I propose to use PTS techniques to identify the transactions that are causing scalability bottlenecks. Using voltage boosting techniques that are possible on Near-Threshold Computing systems (NTC) [51], we show it is possible to construct an asymmetrical system where the accelerator core moves around to accelerate transactions. This eliminates the need to migrate the transaction to another core, instead the core goes to the transaction. I show in this thesis that using voltage boosting and PTS to guide when and where to boost

can further alleviate scalability bottlenecks and further increase performance.

1.4.3 Multi-Threaded Fetch Throttling

The final contribution is to Multi-Threaded (MT) processor fetch policy. One of the problems with MT processors is determining the fetch policy that provides optimal throughput. In transactional applications with scalability bottlenecks, a fair and bottleneck oblivious fetch policy will give bandwidth to threads that are not doing useful work. This effectively reduces the realizable performance in an MT processor with HTM support.

I propose in this thesis to leverage the ideas from the voltage boosting and PTS implementations and apply it to MT fetch policies. The problem with an MT fetch policy is the reverse of the boosting problem. Instead of trying to find the core that is preventing cores from proceeding forward and accelerating that core, the MT fetch policy has to find the thread doing no useful work and slow it down. Finding threads that are not providing forward progress to the transactional program and giving them less fetch bandwidth allows the remaining threads to execute faster. PTS again be used to predict which transactions cannot run, and give hints to the core about reducing their fetch bandwidth. I will show that this is an effective method for reducing scalability bottlenecks in transactional applications in the context of MT processors.

1.5 Organization

This thesis is organized in the following manner. In Chapter 2 I will cover the material necessary to understand the concept of transactional memory and how it is used in this thesis. Chapter 2 will also cover the basics of heterogeneous chip multi-processors and how they will be useful in future systems. Chapter 2 also covers the work that is related to this thesis in the areas of transactional memory and heterogeneous computing as it pertains to eliminating scalability bottlenecks. Chapter 3 covers the motivation, implementation, and evaluation of the "Proactive Transaction Scheduling" technique for transactional memory contention management. Chapter 4 covers the "Bloom Filter Guided Transaction Scheduling" technique, which is an improved version of PTS. Chapter 5 covers the motivation, techniques used to guide core boosting for eliminating scalability bottlenecks, and evaluation of the technique. Chapter 6 covers techniques for guiding MT fetch policies in processors using transactional memory as well as its evaluation. Finally, in Chapter 7 I offer my conclusions and reflections on this thesis and the research that has gone into it.

CHAPTER 2

Background and Related Work

In this chapter I cover the background material necessary to understand the terminology, concepts and challenges present in modern day parallel architecture along with related work in the areas this thesis covers. This chapter first presents a background on basic chip multi-processor architectures and the challenges exposed by them. This chapter then covers the specifics of *Hardware Transactional Memory* (HTM) as it pertains to this thesis as a solution to the challenge of scalable, tractable synchronization for chip multi-processors. It will cover the basic topics on how HTM works, as well as cover more in depth on the specific topic of contention management and the related work in this area. The chapter will conclude with describing challenges and related work in the area of asymmetric parallel architecture thread scheduling and multi-threaded instruction fetch policy.

2.1 Chip Multi-processor Architecture

Chip Multi-processors are an extension of traditional parallel architectures that used multiple processor chips interconnected by external busses. True multi-processor machines have existed since the 1960's with the Burroughs D825 [54] system. Now, because of available transistor budgets, an entire multi-processor system can be placed on one chip.

2.1.1 Symmetric Chip-Multiprocessor

The most common form of CMP that is routinely programmed for and manufactured is the Symmetric Multi-Processor (SMP). SMP machines employ a set of identical cores—identical in performance, area, power and instruction set. Figure 2.1 shows the organization of an SMP system. The SMP system is easy to build and to an extent, write code for. It is easy to program in regards to determining thread schedules. Since every processor is identical, code will run at the same speed on any processor. This removes a variable from the

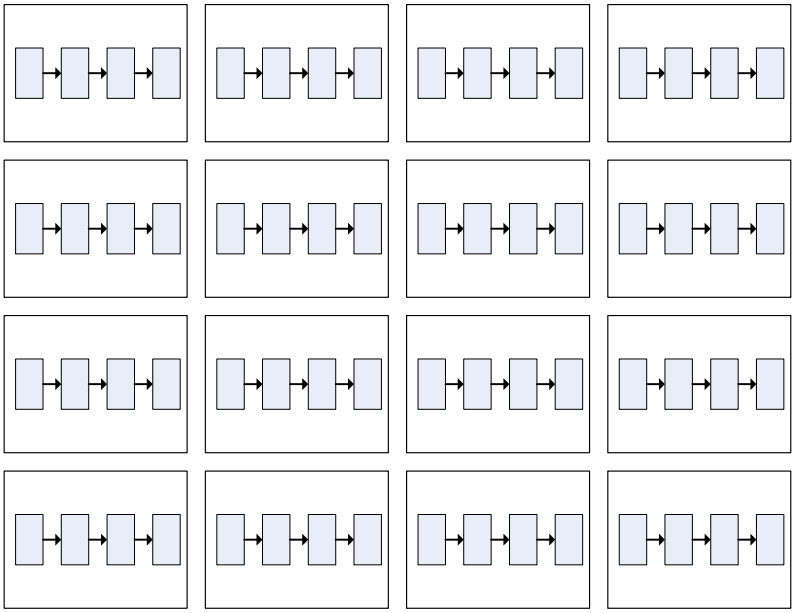


Figure 2.1: An SMP system with multiple identical cores.

thread scheduling problem, which is to determine an optimal schedule to run threads. CMPs that are SMP systems with a large number of cores usually sacrifice single thread performance in an attempt to boost parallel performance such as Sun's Niagara 2 processor [70]. This leads to the main disadvantage of SMP systems which is they deal very poorly with serial code. This is because they lack high single-thread performance cores in favor of identical cores to ease the programming and scheduling burden with the requirement programmers must write efficient parallel code for them. As more and more researchers and developers attempt to extend parallel programming into general domains away from historically good domains (scientific computing, servers), they have found SMPs constrained by serial sections of code, limiting scaling to more cores. This is because it is becoming harder to decompose serial sections into parallel code to take advantage of SMP style CMPs. Codes that have many blocking critical sections that suffer from contention also perform poorly on SMP systems.

2.1.2 Asymmetric Chip-Multiprocessor

Asymmetric Multi-Processors (AMPs) attempt to provide an architecture that can solve the problems with SMPs. As larger CMPs have been proposed it has been troublesome to create applications to take advantage profitably of more than a handful of cores in an SMP system. One of the main reasons for this lack of scaling is due to scalability bottlenecks

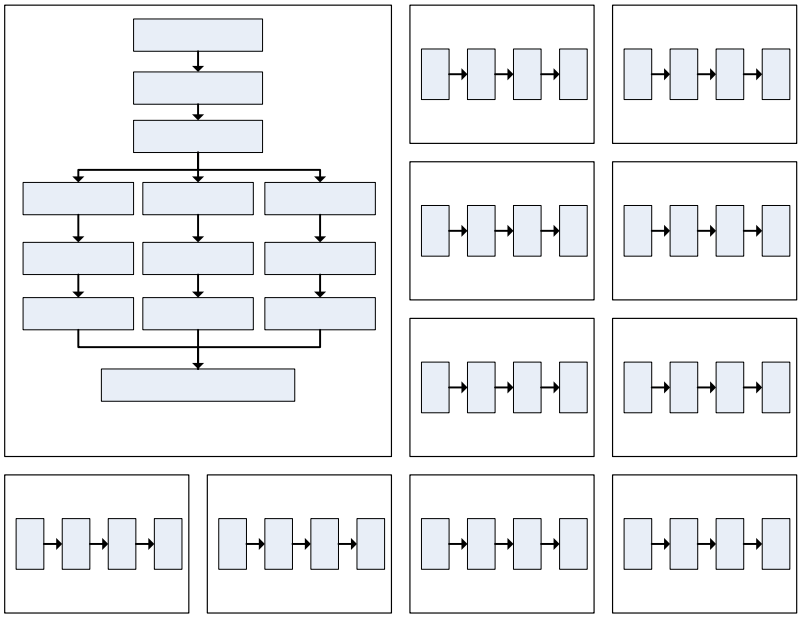


Figure 2.2: An AMP system with many small cores and one large high performance core with the same area as the SMP system.

caused by small serial sections of code. Hill and Marty [68] showed in their analysis of CMPs that the serial portion of code severely limited scaling when using SMP systems. To solve these bottlenecks, they proposed using AMPs—CMPs with a mix of cores with differing single-thread performance, area and power—to reduce these bottlenecks. AMPs can accelerate the serial sections of code on the fast cores, making the serial section be less of the execution. Work by Eyerman et al. [56] looked at modeling the effect of critical sections serializing code mathematically in a similar fashion to the work by Hill and Marty [68]. They also come to the conclusion that AMPs are a better design decision than SMPs. AMPs are not a new idea, first being proposed by Kumar et al. [73] for reaching the optimal energy and performance point for single-threaded applications. They are now being looked at in a new light to allow for continued scaling for programs that are not easily parallelized.

Figure 2.2 shows a depiction of one type of AMP system. In this figure, the AMP has some small cores, and one large high performance core, depicted by its larger area and multiple pipelines. AMPs can use multiple methods to attain cores with different performance characteristics. Using a mix of big cores and little cores is one such way. Another proposal is to use an SMP system and allow cores to run at their optimal voltage and frequency so some cores clock faster or slower than others due to variation in manufacturing. A proposal by Dreslinski [52] presents an AMP architecture that is dynamic instead of fixed at design

time by utilizing voltage boosting. AMPs can also be built from a mix of programmable accelerators to provide different compute capabilities that match specific applications. This is being done by AMD with their Fusion cores [6] by integrating a GPU and CPU on the same die where the different processors have different ISAs.

2.1.3 Per Core Multi-threading

As architects move towards giving more parallel resources to programmers in the form of more cores, another problem emerges. This problem is under used execution resources in the parallel cores due to pipeline stalls from long latency memory accesses. These long latency accesses become more common in CMPs because memory resources are shared. As more cores are added to a CMP the less memory resources it has access to. To combat under utilization, Multi-Threading (MT) can be used. MT processors are processor cores that support the execution of multiple software threads concurrently on the processor hardware to more efficiently share resources and keep utilization and throughput high in the presence of high latency operations, such as cache misses. To enable MT, a processor core needs to replicate the needed control structures to allow multiple software threads to exist in hardware on the same core. Traditionally only one thread of execution runs on a core at a given time. The replicated structures are usually the register file and control registers, the remaining processor units can be shared by hardware threads. There are three main implementations of MT cores. Coarse-grained MT forces a context switch when a high latency operation is encountered. A context switch means one hardware thread is suspended and another is allowed to execute. In MT processors, only one thread is usually allowed to issue instructions at a time. An early proposal of coarse-grained MT was the APRIL processor by Agarwal et al. [14]. Fine-grained MT context switches each cycle. The Tera Computer by Alverson et al. [17] was an early proposal that used fine-grained MT. The Niagara processor from Sun Microsystems [71] uses fine-grained MT as well. The last type is Simultaneous MT (SMT), first proposed by Tullsen et al. [108]. SMT allows a processor to dispatch instructions from multiple threads in a cycle at a time. This allows better utilization of resources over coarse and fine-grained MT because it removes the requirement for only one thread to be issuing at a time.

To software, an MT processor makes multiple logical processors visible per physical processor. This gives software access to more concurrently executing threads, potentially increasing performance. Many current CMPs now use MT to enable better overall throughput. Examples include the Sun Microsystems Niagra 1 [71], Sun Microsystems Niagara 2 [70], Intel Xeon [8], and Nvidia Fermi architecture [9].

2.2 Chip Multi-processor Challenges

The previous section introduced current architectures for CMPs that expose many threads to a programmer to leverage. This has lead to many challenges in profitably leveraging these programmer exposed hardware resources. This section will cover the challenges of thread synchronization, AMP thread scheduling and resource partitioning in MT cores.

2.2.1 Thread Synchronization

As introduced in Chapter 1, scalable and efficient synchronization among threads in a CMP is a hard challenge. The traditional methods of synchronization do not scale well as CMPs get larger and the pool of programmers that must program CMPs gets larger as CMPs continue to enter more computing domains. This is due to traditional methods requiring explicit control by the programmer to effect an efficient and scalable program. As argued previously, requiring programmers to perform such explicit control is not desirable and new methods must be investigated to allow easier thread synchronization for shared memory CMPs. In section 2.3 I will introduce Hardware Transactional Memory as an architecture technique to enable efficient and scalable synchronization. I will also introduce contention management, an automatic technique that aims to provide acceptable performance with little to no programmer intervention.

2.2.2 Thread Scheduling in AMPs

AMPs raise many interesting problems due to their heterogeneous nature. The biggest problem is thread scheduling. Programs themselves need to be aware of the asymmetry to direct the Operating System (OS) and place the correct threads on the correct cores to reduce scalability bottlenecks and attain the best performance. Alternatively the OS needs to deal with the asymmetry when it is scheduling threads and intelligently devise a schedule that will make best use of the asymmetric cores. Work by Lakshminarayana et al. [76], Balakrishnan et al. [23] and Li et al. [80] are examples of looking at scheduling threads on an AMP from the OS and program perspective. The dynamic and phasic nature of most programs makes this scheduling problem hard. It is difficult to deduce a static schedule during programming or compilation, and the OS schedules at too coarse a granularity in most cases. Another interesting problem to the software community is AMPs may leverage multiple ISAs. Writing programs to take advantage of such a situation as done by Nightingale et al. [86] is part of ongoing research. If AMPs can be leveraged properly, they can reduce the impact of serial sections of code and improve overall scaling for parallel programs, lessening the burden of trying to decompose hard to parallelize codes by

using asymmetric resources. In section 2.6 I will present related work in the area of thread scheduling in AMPs from an architectural standpoint and how it pertains to this thesis in the context of scalability bottlenecks.

2.2.3 Resource Partitioning in Multi-Threaded Cores

One key area of research in MT enabled processors is looking at different fetch policies in MT processors to maximize resource utilization and per thread performance. The optimal fetch policy has been to found to be different per application and no policy wins in all cases. This is a challenge because the system must now dynamically find the optimal policy, just as with thread scheduling to AMPs. In section 2.7 the related work in the area of MT fetch policies and how they pertain to this thesis.

2.3 Hardware Transactional Memory

To enable fast, efficient and tractable synchronization, transactional programming through the use of Transactional Memory has been proposed. As discussed in Chapter 1 there are three methods for building TM: Hardware, Software and Hybrid. I cover only the mechanisms of Hardware Transactional Memory (HTM) here, as Software Transactional Memory (STM) and Hybrid Transactional Memories (HyTM) are not applicable to this thesis. Readers interested in these two methods of implementing TM as well as more exhaustive descriptions of the topics covered in this section should refer to the two books that focus exclusively on presenting a survey of the state-of-the-art in Transactional Memory by Harris et al. [65] and Larus and Rajwar [77].

Hardware Transactional Memory uses architectural techniques to provide high performance transactional programming semantics to the programmer. This is done by adding to the Instruction Set Architecture (ISA) to enable the construction of transactional programs and expose transactions to the architecture. The memory system is modified to provide conflict detection and memory versioning support. Additional support for nesting, non-transactional operations like Input/Output (I/O) and contention management will also be covered in this section.

The ISA support for HTM must consist of at least the following two instructions: `TM_BEGIN` and `TM_END`. The `TM_BEGIN` instruction is used to transition the processor and attached caches into transactional mode. Entering into transactional mode consists of primarily taking a register checkpoint that the processor can revert to in case the need to abort the transaction and restart arises. All accessed memory addresses after the `TM_BEGIN` instruction are protected by the hardware and made to be viewed as atomic to remote processors.

The TM_END instruction causes the processor to commit all accessed memory addresses as one transactional unit. If in between the TM_BEGIN and TM_END instruction, a condition is detected that would break the atomic and isolated conditions required of TM, the cpu will automatically retry the transaction by reverting to the register checkpoint and restoring memory to its proper state before the TM_BEGIN instruction was executed. In effect the transaction appears as if it never executed when a retry is needed in keeping with transactional semantics. Other instructions can be added to allow for more flexible and richer transactional support. Such instructions include the TM_ABORT instruction that allows the programmer to abort transactions whether or not a conflict was detected. This is useful for certain programs, such as the transactional version [109] of Lee's routing algorithm [78]. Another instruction that is supported by some HTMs is the TM_RELEASE instruction. This instruction allows the TM system to remove transactional protection from accessed memory locations. This is useful in lowering contention as investigated by Skare et al. [101] and Christie et al. [44]. Other additional instructions have been proposed to fit the needs of the authors and providing transactional programming support they were trying to provide in the following works [33, 82, 94].

Conflict detection is the process by which the TM system determines if two or more transactions are trying to make modifications to the same memory locations in a fashion that is not serializable. Serializability is the property by which a specific ordering can be assigned to accesses (access 1 happens before access 2 which happens before access 3 etc.) that does not include any loops in the happens before graph. A loop would be if the following access pattern was observed: access 1 happens before access 2 which happens before access 1. HTM's detect this condition by modifying the coherence protocol and cache lines with extra *"Transactional"* states to detect conflicting accesses to the same memory address. These conflicting accesses would be Read after Write conflicts, Write after Write conflicts and Write after Read conflicts. Read after Reads are ignored by HTMs as they do not constitute a conflict. There are two ways in which conflict detection can performed, it can be performed as *Eager* or *Lazy*. Eager conflict detection continuously watches the transactional execution looking for conflicting accesses. Upon the first instance of a conflicting access, an Eager conflict detection mechanism will decide which processor making the conflicting access to abort. Because this method of conflict detection resolves conflicts early, it has the ability to allow a minimum of wasted work to occur within transactions. But pathologies can occur, such as cascading aborts as seen by Bobba et al. [36]. The other method of conflict detection is Lazy conflict detection. Lazy conflict detection defers detecting conflicting accesses until commit (TM_END instruction). Upon commit, the HTM will attempt to validate the memory addresses read and written (called the Read/Write Set,

or RWSet). If other committed transactions have already made changes to locations in the validating transactions RWSet, the transaction aborts. Lazy conflict detection can lead to more wasted work from deferring conflict detection. On the other hand it can allow more parallelism as seen by Tomic et al. [106]. The work by Tomic et al. additionally devised methods to mix eager and lazy conflict detection in an HTM.

HTMs also have two methods of versioning memory locations touched by a transaction. Because TM has to provide the semantics of an all or nothing transaction, it has to keep two versions of the memory while executing. It must keep the old version of memory that is restored on abort and a new version that will be seen after commit. The two methods of versioning are also termed *Eager* and *Lazy*. Eager versioning places the new version of the memory location in the place where it will be stored on commit. Therefore the old version must be stored to the side, and restored on commit. There are multiple ways to accomplish this, one method is store the old version in a log in virtual memory, and restoration happens on abort by the way of a software routine that reads the log and overwrites the new memory value with the old value. This was first proposed by Moore et al. [84]. Another method is to force the upper level caches to write back old values to lower level caches and store the new value in the upper level caches. On abort the upper level caches are flash invalidated as proposed by Blundell et al. [33]. Eager versioning allows for very fast commits, while aborts can be slow in the case of LogTM. Lazy versioning, as with lazy conflict detection waits until commit to place the new values into their final locations. Lazy versioning needs a mechanism to store the new values off to the side that can be accessed later during commit. The very first HTM, proposed by Herlihy and Moss [67], used lazy versioning and did so by dividing the cache into two pieces to hold new transactional state and old transactional state. On commit the pointers denoting which half held old values and which held new values were flipped on commit. Other implementations such as Transactional Coherence and Consistency (TCC) HTM by Hammond et al. [64] and AMD's Advanced Synchronization Facility (ASF) [44] propose using special associative write buffers to hold modified lines until commit. Lazy versioning allows for fast aborts as the old data still remains in place. Commits, on the other hand, can be slow as the caches must broadcast all stores out to the remote processors one at a time while holding a global commit lock which can limit performance. There are ways to remove the need for a global commit lock. One proposal is from Chafi et al. [43] where transactions commit the write set in sorted order to prevent deadlock. Using conflict detection and memory versioning, a taxonomy of HTM proposals can be constructed. Table 2.1 shows a selection of HTM proposals and how they fit. As can be seen the majority of proposals fit in either the Lazy/Lazy or Eager/Eager block in the taxonomy and are all academic works. This is due to these two design points

having desirable properties. These properties include exposing higher amounts of parallelism by default in Lazy/Lazy designs or limiting wasted work in Eager/Eager designs. These design points also offer interesting system redesign opportunities in terms of coherence protocols and cache design. The third design point, Eager conflict detection and Lazy versioning has less examples as it was discovered to have a large problem with contention as seen in the paper by Bobba et al. [36] cataloguing pathologies. Interestingly enough, two commercial companies Sun and AMD designed HTMs in this space. This was chosen as the Eager/Lazy design requires the fewest changes to an existing chip-multiprocessor's memory system. Most importantly, and Eager/Lazy HTM can be designed in such a way that it leaves cache coherence completely untouched. This is important to companies as coherence changes require massive efforts to debug, because bugs in the coherence scheme can prevent a system from functioning.

		Version Management	
		Lazy	Eager
Conflict Detection	Lazy	TCC [64] Bulk [42] RTM [100] FlexTM [99] Colorama [41] EazyTM [106]	
	Eager	SigTM [38] MIT LTM [18] Intel/Brown VTM [90] AMD ASF [44] Sun ROCK [48]	LogTM [84] MIT UTM [18] MetaTM [92] TokenTM [35] DATM [91]

Table 2.1: Taxonomy of select Hardware Transactional Memory implementations.

HTM also needs to support nesting of transactions and non-transactional actions inside of a transaction (notably Input/Output). Nesting is when a transaction contains transactions. Support for nesting is useful for when programmers want to compose programs using transactions that themselves use libraries or functions that use transactions internally. With traditional synchronization, composing critical sections that contain other critical sections can be non-trivial and difficult. There are three types of nesting that have been proposed: closed, open and parallel nesting. Closed nesting is the most intuitive to understand, as the parent transaction subsumes any child transactions and all child transactions must commit successfully for the parent transaction to also commit. Closed nesting can be implemented with flat semantics, where child transaction `TM_BEGIN` and `TM_END` are treated as NOPS

and only the parent `TM_BEGIN` and `TM_END` are processed. Closed nesting can also allow partial aborts, where an abort only rolls back the child transaction that is conflicting, and restarts from the `TM_BEGIN` of the conflicting child transaction. The implementation details needed to support this type of nesting for an Eager/Eager HTM can be found in work by Moravan et al. [85]. Open nesting is a more complicated form of nesting that allows child transactions to commit early and allow their changes to memory to be visible before the parent transaction completes. The complications come from when a parent transaction of an already committed child transaction is aborted. The child transaction has already been committed, and the HTM system is no longer able to roll back the changes. To solve this, McDonald et al. [82] proposed using special abort handlers for open nested child transactions to provide the necessary fix-up operations to ensure correct operation when the parent transaction restarts. These fix-up operations must be programmer supplied. The last type of nesting is called parallel nesting [21, 24, 15]. Parallel nesting is where a parent transaction spawns additional threads that themselves contain transactions. This presents numerous complications with versioning and conflict detection that are not be covered here.

Supporting non-transactional operations is one of the remaining challenges in HTM design. This is also true of all transactional systems, including databases as stated by Gray and Rueter [59]. The main problem with non-transactional actions is that they break atomicity and isolation guarantees, and in many cases cannot be rolled back. There have been many proposals for solving this problem, but no universally accepted solution has yet emerged in the HTM community, and therefore is still an active area of research. Buffering non-transactional actions until commit has been proposed [82]. This works for some non-transactional actions, unless a transaction both writes and reads, for example writing to the terminal displaying a prompt and waiting for input. The transaction will hang because the write will not be visible for the user to input data to the terminal. Another method to support non-transactional actions and make them appear transactional is to require compensating actions in case the transaction aborts. This also has problems because it requires the programmer to define what could be complex compensation actions for the transaction. On top of these compensations there are non-transactional actions that can not be compensated for, such as printer output. Another proposal is to grant permission to one transaction at a time in the system to perform non-transactional actions [33]. If permission is granted to a transaction to perform non-transactional actions it is guaranteed to commit, therefore eliminating the rollback problem. A problem noted by researchers with this approach is that parallelism is limited if more than one transaction wants to perform non-transactional actions and they are independent.

2.4 Transactional Memory Benchmarks

Transactional Memory benchmarks are important for the evaluation of HTM designs. Initially, transactional benchmarks did not exist. Researchers were forced to rely on already parallel applications and simply replace the locks in these programs with transactions. A popular application set for early HTM work was the SPLASH2 [110] benchmark suite. It featured scientific applications that used small amounts of synchronization. The transactions in this benchmark suite were small, and contention was very infrequent. This led researchers to believe that TM in general did not have to worry about contention in the common case. However, these benchmarks were very highly tuned applications from experts, meant to scale to a large number of cores and even across machines. Therefore synchronization was kept to a minimum.

Because HTM was meant to enable a higher level of abstraction of transactional programming to programmers, the likelihood that future transactional applications would feature small, unlikely to conflict critical sections like those in SPLASH2 is small. This led researchers to develop new benchmarks designed to represent applications that HTM systems were likely to encounter. Specifically applications with large, coarse grained transactions that were likely to have moderate to high contention. Such benchmarks include the STAMP [37] benchmark suite, which does showcase large transactions that also has a high amount of contention. Other benchmarks emerged such as a transactional version of a parallel Delaunay triangulation by Kulkarni et al. [72] and a transactional implementation of Lee's routing algorithm [78] by Watson et al. [109]. Other transactional benchmarks have been developed, such as versions of the Linux kernel using transactions for synchronization instead of locks. The first implementation was a transactional version of the v2.6 kernel by Rossbach et al. [94]. It had small critical sections and was more akin to the SPLASH2 benchmarks but showcased that highly complex parallel programs such as the Linux could use transactions. Later work by Hoffman et al. [69] developed a transactional version of the v2.4 Linux kernel which used large transactions and showed that transactions allowed coarse grained semantics with performance approaching fine grained locks. These newer benchmarks show that attaining scalable performance and not requiring expert programmers is a large challenge. In this thesis I use the STAMP benchmark suite to evaluate my solutions to the scalability problem.

2.5 Contention Management for Transactional Memories

This thesis deals very heavily with contention management in transactional memory, as conflicts in TM form scalability bottlenecks that can induce pathological performance degradation. As seen in Tables 1.1 and 1.2 in Chapter 1 for the STAMP benchmark suite, conflicts in HTM can lead to less than serial performance in cases. Because of this, contention management techniques have been a large area of research for all in the TM research community. One of the first works that realized the importance of advanced contention management and concentrated on identifying contention pathologies was written by Bobba et al. [36]. Bobba et al. found that Eager/Eager and Eager/Lazy HTMs suffered from contention more than Lazy/Lazy.This has led to multiple ways to deal with contention to allow HTM provide the fast and efficient transactions.

2.5.1 Programmer Managed

The most powerful and flexible method for reducing contention is to have the programmer tune transactions to guarantee low contention. There are four types of programmer guided contention management: minimize transaction size, selective cache line marking, open-nesting and early release.

Minimizing transaction size, and breaking large transactions into multiple small transactions can reduce contention. This requires the programmer to look at coarse grained transactions that may contain functions and determine if it is safe to make the transaction smaller and/or break the transaction into multiple smaller transactions. It has been shown that smaller transactions do reduce contention as it minimizes the window of time in which transactions can conflict in. On the other hand, making fine grained transactions to reduce contention goes against the aim of transactional programming which is: coarse-grained semantics are acceptable regardless of the transactional implementation underneath.

Selective marking of individual data elements that should be protected is one method to reduce contention. The program, or compiler if possible, must inspect each transaction and mark only the necessary lines that must be tracked by the TM system. This keeps the RWSet small and reduces the risk of accidental conflict due to false sharing in the caches. For STMs, this is more important as STMs have problems with increasing overheads as the number of locations that must be tracked grows. Because of this, selective marking is commonly seen and practiced by STM implementations and benchmarks. For HTMs, tracking overhead is negligible to performance in most cases but increased chances of contention can reduce performance seen. An example of how selective marking can be used to make a minimal transaction RWSet is shown in Example 2.1 for inserting into a sorted

linked list. As shown in the example it takes some clever manipulations to reduce conflict opportunities. The examples checks to make sure nothing changed between finding the insertion point and inserting the new list node transactionally. If a change to the insertion point occurred, then the transaction needs to be aborted by the user assuming user abort is supported by the HTM's ISA.

Example 2.1 Using Selective Marking to optimize out contention for sorted linked list insert.

```
1  void* threadwork()
2  {
3    ...
4    TM_BEGIN;
5    insertIntoList(5);
6    TM_END;
7    ...
8  }
9  // example is simplified not to show
10 // the corner cases of inserting at
11 // the head or tail of the list
12 void insertIntoList(int val)
13 {
14   ptr = head;
15   while (ptr->val < val) {
16     ptr = ptr->next;
17   }
18   TM_READ(ptr = ptr->prev);
19   TM_READ(next = *ptr->next);
20   if (*ptr->val > val) TX_ABORT;
21   else {
22     node = newNode(val);
23     TM_WRITE(*ptr->next = node);
24     TM_WRITE(next->prev = node);
25   }
26 }
```

Open nesting is another option that can be used to limit the size of the transaction's RWSet and therefore limit contention. Example 2.2 shows how open nesting can be used to accomplish RWSet minimization as was done in the previous example. Again, this example is more complicated than the simple semantics promised by transactional programming in Chapter 1 of placing `TX_BEGIN` and `TX_END` around the section of code that needs to made atomic (in this example, the function call `insertIntoList`). Instead the programmer in this example is making low level optimizations similar to scalable locking algorithms for linked list insert. The algorithm is very similar to the previous example, find the location

to insert in, and check if the insert can be made or abort and retry. In Example 2.2 no compensation actions are required. But, as described in the work by McDonald et al. [82], open nested transactions may require non-trivial compensation actions to repair program state to a consistent before transaction execution state if the parent transaction aborts.

Example 2.2 Using open nesting to optimize out contention for sorted linked list insert.

```
1  void* threadwork()
2  {
3    ...
4    TM_BEGIN;
5    insertIntoList(5);
6    TM_END;
7    ...
8  }
9  // example is simplified not to show
10 // the corner cases of inserting at
11 // the head or tail of the list
12 void insertIntoList(int val)
13 {
14   TM_BEGIN_OPEN(compensationAction);
15   ptr = head;
16   while (ptr->val < val) {
17     ptr = ptr->next;
18   }
19   TM_END_OPEN;
20   ptr = ptr->prev;
21   next = *ptr->next;
22   if (*ptr->val > val) TX_ABORT;
23   else {
24     node = newNode(val);
25     *ptr->next = node;
26     next->prev = node;
27   }
28 }
29
30 void compensationAction()
31 { return; }
```

The last technique to limit transaction size is to use early release as proposed by Skare and Kozyrakis [101]. Early release uses a new instruction in HTM to signal the caches to release from the RWSet the passed in address. This allows for dynamic trimming of the transaction size in a similar fashion to open-nesting, except that early release is more limited in that is does not assume the presence of compensation actions. The programmer or compiler must know releasing the cache line early from the RWSet is safe. An exam-

ple of early release is shown in Example 2.3. As can be seen, this example algorithm is slightly simpler to the previous two. As the transaction scans the list, it pulls two nodes into the RWSet and releases the oldest third node. This eliminates the need for a `TX_ABORT` instruction and check code before inserting as in the previous examples.

Example 2.3 Using early release to optimize out contention for a sorted linked list insert.

```
1  void* threadwork()
2  {
3    ...
4    TM_BEGIN;
5    insertIntoList(5);
6    TM_END;
7    ...
8  }
9  // example is simplified not to show
10 // the corner cases of inserting at
11 // the head or tail of the list
12 void insertIntoList(int val)
13 {
14   ptr = head;
15   prev = head;
16   oprev = head;
17   while (ptr->val < val) {
18     oprev = prev;
19     prev = ptr;
20     ptr = ptr->next;
21     TM_RELEASE(oprev); //free prev->prev from RWSet
22   }
23   node = newNode(val);
24   prev->next = node;
25   ptr->prev = node;
26 }
```

As seen in the examples, requiring the programmer to manually tune transaction RWSets can be complicated (building a compiler to understand how to apply these optimizations would also be problematic for this example as it involves pointer chasing). Requiring such optimizations would negate the main claim that transactional programming supported by TM is easier to use. While having this support built into an HTM would be beneficial for expert programmers and optimizing compilers, more transparent solutions that offer acceptable performance should be pursued.

Programmer managed approaches do lead to low contention because it reduces the window of contention by relaxing the atomicity and isolation constraints of the HTM. It also

eliminates many of the advantages of transactional programming, specifically eliminating the abstraction because the underlying HTM system must be tuned for. By requiring the programmer to hand inspect transactions to avoid contention it makes transactional programming no easier than traditional synchronization. Programmer independent methods must be devised as it cannot be expected of programmers to tune their programs.

2.5.2 Contention Managers

Contention managers attempt to alleviate pathological conflicts in TM and ensure acceptable forward progress transparently to the programmer. In HTMs, contention managers can be designed, depending on complexity, either in hardware or software. Such automatic management is enabled because HTM exposes the transaction as a primitive to the architecture. This allows the architects to devise systems that can reason about critical section behavior and implement strategies to alleviate contention and increase performance. Research into transparent contention management has yielded three types of contention managers: reactive, proactive and data forwarding/predicting.

2.5.2.1 Reactive Contention Managers

Reactive contention managers operate only when transactions conflict and need to be serialized. They react to conflicts. Because reactive managers are only invoked when conflicts are detected, they have zero overhead when no conflicts exist. Reactive managers can still be arbitrarily complex in their conflict resolution scheme when invoked. The main drawback of a reactive contention manager is that it assumes conflicts are rare events. If conflicts keep repeating, performance can be degraded when using reactive contention management because there is no memory of previous conflicts. Reactive managers usually use some type of backoff to stall conflicting transactions to clear the conflict. An example of reactive contention management is shown in Figure 2.3. As seen in the figure, the conflict can happen repeatedly as the backoff based reactive manager makes guesses as to how long to delay the conflicting transaction to clear the conflict. As seen in Figure 2.3, reactive managers can waste execution time and resources doing backoff.

Reactive contention managers are the simplest type of contention manager, and the first to be used and developed for HTMs. Early HTMs such as LogTM [84] and TCC [64] used very simple reactive managers because their benchmarks indicated that contention was not an issue. In STM research, contention was a problem as it exacerbated performance issues due to high overheads of STM implementations. Therefore work by Scherer et al. [96, 97] studied many reactive contention managers. Scherer et al. experimented with multiple

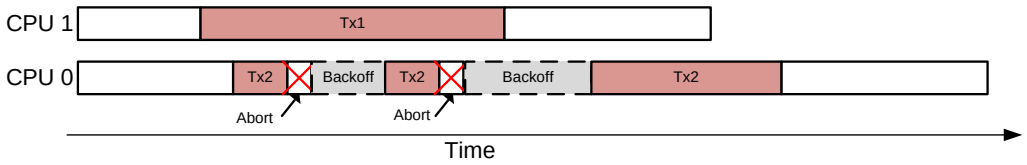


Figure 2.3: Reactive contention manager operation.

policies that tried to balance various heuristics to decide what transactions should abort when a conflict was detected. These heuristics varied from simply always aborting the requesting transaction, giving priority to the transaction that owns the memory location, to complex managers that looked at how many times a transaction had aborted in the past, how much work it had done, how long it had waited to execute successfully among others. The major conclusion from both works was that no set of heuristics was the best. Scherer et al. came to the conclusion that reactive managers must be tailored to the workload to arrive at the best contention management scheme. Other reactive managers used stalling and suspending of conflicting transactions to manage conflicting transactions such as work by Zilles and Baugh [115]. This is similar to the ideas that will be presented in the following chapters, but it is still a reactive management scheme.

2.5.2.2 Proactive Contention Management

Proactive contention managers are different from reactive managers in that they attempt to avoid contention before it occurs. Because proactive managers try to avoid contention, they approach contention management as a scheduling problem. Proactive contention managers assume conflicts to be frequent and re-occurring. This is a valid assumption as future transactional applications, written by non-expert programmers will be more likely to contain less optimal transaction construction. Proactive managers do have drawbacks. The proactive contention manager operates even in the presence of no contention, therefore it incurs higher overheads over a reactive manager. This makes a proactive manager sub-optimal in low contention situations. On the other hand, proactive managers typically perform very well when contention is high. Figure 2.4 shows how a proactive manager operates. It uses some execution time at the beginning of each transaction to determine what transaction it should run next to keep contention low. This can increase performance if the up-front cost of scheduling eliminates wasted work due to contention.

Proactive scheduling can find its roots in database research. Proactive scheduling solutions, such as work by Victor et al. [79] were used to predict and schedule around contention in small real time databases. Scaling to larger databases with large arbitrary transactions

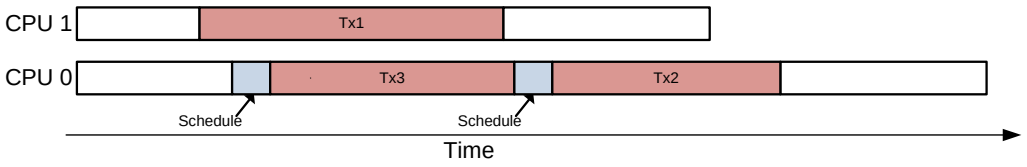


Figure 2.4: Proactive contention manager operation.

was prohibitive for the technique and such scheduling techniques remain limited in this domain. As transactional memory research has grown, researchers are looking down similar paths originally investigated by database researchers, spawning work that has been both empirical and theoretical.

The first proactive manager was proposed by Bai et al. [22]. In Bai et al.'s solution, before every transaction begin, a look-up table was consulted for a key to determine if the transaction should execute with the already executing transactions in the system. They found this worked very well. The main draw-back to this work is the look-up table had to be constructed a priori specifically for each benchmark. This meant that the technique could not be used for arbitrary applications.

Dolev et al. [49] proposed a proactive technique called CAR-STM for Software Transactional Memories that placed transactions on queues behind transactions it conflicted with in the past. These conflict relations persisted throughout the program execution therefore preventing pairs of transactions from ever conflicting again in the future. This lead to overly pessimistic contention management. In some cases CAR-STM could completely serialize execution of transactions in high contention applications. A solution proposed by Dolev et al. to correct this pessimistic scheduling was to allow the programmer to provide hints to the STM system about the chances of future conflicts between pairs of transactions. Like the work by Bai et al. this requires knowledge by the programmer to be provided to the system to increase performance. Therefore the technique is limited.

Ansari et al. proposed two proactive contention managers for STMs. The idea proposed in [19] used commit rate to determine when to schedule more transactions concurrently. If the commit rate dropped below a watermark, then the number of concurrent transactions was reduced. When commit rate increased more threads were allowed to execute concurrently. Ansari et al. [20] also proposed an idea similar to Dolev et al. On conflict, the transaction would queue behind the transaction that it conflicted with and remain there to prevent repeated conflicts. As with CAR-STM it can also suffer overly pessimistic scheduling.

Yoo and Lee [113] proposed a proactive scheduling technique called "Adaptive Trans-

action Scheduling" (ATS) that is similar to that proposed by Ansari et al. It uses abort rate to devise a heuristic called conflict pressure which is a measure of instantaneous conflict rate. Conflict pressure is used to throttle execution of concurrent transactions. When the pressure is high, ATS throttles execution to be close to serial to guarantee forward progress. When pressure is low ATS allows highly concurrent execution. Yoo and Lee's proposal fits well with HTMs as it is very low overhead, and therefore sees good performance. Still, it does have the potential to schedule too pessimistically as well. As can be seen most of the work presented so far dealing with proactive scheduling has been very simple. The proposals have used simple techniques to affect schedules, or require programmer input to devise of a suitable schedule.

Dragojević et al.'s [50] version of proactive scheduling uses prediction of future read and write sets to determine whether transactions can proceed in parallel with other currently running transactions. Before every transaction begin, Dragojević et al. predict the read and write set of the transaction wishing to execute by using past history of memory accesses. They then check this prediction against the current RWSets of all running transactions. This is very high overhead, and as such is only implementable in STM. The technique does show performance gains using this method. This method is very similar to the proposals presented in this thesis. I also use prediction to schedule transactions around future conflicts, but do so in a lighter weight fashion.

Proactive scheduling has been receiving attention from many types of researchers, even theoretical research into TM scheduling to prove algorithmic soundness is being conducted. Work by Guerraoui et al. [61] was the first to approach the problem in this fashion. This has led others to propose new contention managers that can be implemented and have provable contention management characteristics, such as the proposal by Sharma et al. [98]. As can be seen, research in contention managers that proactively schedule transactions to avoid contention is a very active research area. This thesis makes multiple contributions to the area of proactive contention management in Chapters 3 and 4.

2.5.2.3 Data Forwarding/Predicting Managers

Data forwarding/predicting techniques are a final transparent technique to alleviate contention. Instead of determining which transaction to abort when a conflict occurs or predicting transaction execution schedules, these techniques allow the conflict to persist and attempt to fix the conflict so both transactions can complete at commit. This is accomplished in the following ways: forwarding modified values from one transaction to the conflicting transactions, or predict the value of the conflicting access to allow continued execution. Commits of conflicted transactions are allowed to proceed if serializability can

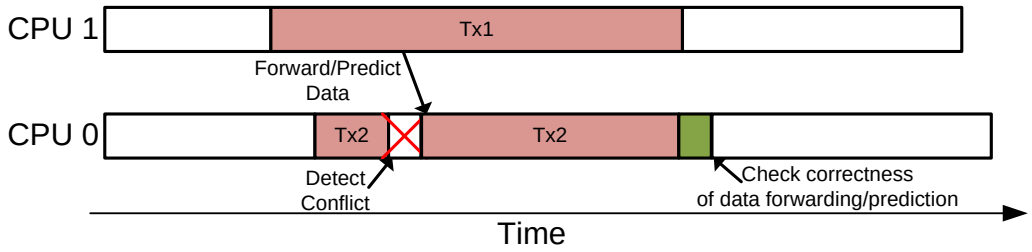


Figure 2.5: Data forwarding/predicting contention manager operation.

be proven between conflicting transactions. This means values that are forwarded can not be changed by the forwarding transaction after being forwarded and value predictions must be correct. Figure 2.5 shows how a data forwarding mechanism would work. When a conflict is detected, the transaction continues with a forwarded or predicted value and checks at commit whether the execution was valid. Approaching contention management by relaxing the isolation constraints in this way reduces contention for a specific class of conflicts called ancillary data conflicts as described by Click [46]. An example of such a conflict would be an insert into a hash table, then incrementing a single common occupancy counter. This type of conflict can cause a parallel operation to be serial among multiple threads because of this single point of contention. Researchers have begun to look for programmer transparent solutions to these types of HTM conflict patterns that cannot be solved in a scalable manner by either reactive or proactive contention management.

The first proposal that attempts to solve these types of ancillary conflicts was proposed by Ramadan et al. [91] called Dependence Aware Transactional Memory (DATM). DATM used very complex cache coherence modifications along with multiple hardware structures to track when a transaction forwarded a data value it modified to another transaction that is causing a potential conflict. Ramadan et al. found that by forwarding data among transactions reduced conflicts sharply and allowed much greater throughput. Tracking this forwarding could become complex as forwarding among transactions could be arbitrarily deep, which made rollback complicated, because the hardware had to find all transactions dependent on the aborting transaction and roll them back too. The cases that caused an abort also became more complicated. If the forwarding transaction happened to modify the forwarded line more than once it would cause dependent transactions to abort. Aborts were also caused when dependency loops were formed, i.e. a transaction T_1 could not forward to T_2 and then have T_2 forward to T_1. The authors did show that the technique worked well for all benchmarks and that the worst case rollback was rare, but the hardware to handle this case was still required. This was the main disadvantage to the DATM proposal, its

complicated hardware to track dependencies and major coherence changes.

A proposal by Blundell et al. [34] called RetCon also tackled the problem of ancillary updates that cause what can be considered false conflicts. Instead of making large changes to the coherence scheme and adding tracking data structures to the memory system to forward data as in DATM, RetCon used additions to the processor to do fix-up operations on data that was determined to have been in a conflict state and also predicted to be ancillary data updates. It did this by first predicting which memory updates could be fixed up after commit in case of conflict. In essence RetCon predicts locations that will not affect transactional execution. It then used hardware structures to track operations done to the memory location, such as adds, subtracts etc. It also recorded boundary conditions that need to be enforced in case the memory was used to guide branching behavior. After commit the processor would read the hardware structures to replay the operations on the data and fix it to be the value it should be as if the conflict had never happened. If the fix-up cannot proceed due to violating boundary conditions, then the transaction is ultimately aborted. Again the main disadvantage to this work is the same as DATM, it requires very complicated hardware to record memory accesses that need fix-up and the operations that need to be applied on replay for fixing.

Others have proposed less aggressive methods to fixing the ancillary data update problem that causes many conflicts. These proposals from Tabba et al. [105] and Pant et al. [87, 88] propose using value prediction instead of fix-up or data forwarding. When a data location is detected to cause a conflict, its future value is predicted and used to allow execution to continue. When the transaction finishes, it checks its prediction before allowing the transaction to commit. These proposals have also shown to work at fixing these types of conflicts, although they are not quite as effective as the more complicated proposals presented above. Overall, data-forwarding contention management is very powerful as it allows the maximum amount of parallelism between transactions to be realized transparently to the programmer. On the other hand, for HTMs it incurs a very high hardware overhead in terms of complexity of implementation.

2.6 Asymmetric Multi-Processor Thread Scheduling

There has been recent interest in heterogenous architectures in industry and academia as the problem of scalability bottlenecks have become more apparent. As discussed previously the main problem is locating threads on the appropriate cores to get the best possible speedup. This has prompted architects to propose architectures that provide hooks to the software to more accurately determine when to move threads in a fine-grained manner i.e.

many thread moves within an OS scheduling quantum. This has been deemed necessary because programs showcase bottlenecks that may be too small to handle at the granularity of OS scheduling, and input dependent in such a way that static scheduling is not possible.

A proposal by Suleman et al. [103] called Accelerated Critical Sections (ACS), presents a CMP design that has one large out-of-order core for serial code with multiple small cores used for the parallel sections. Suleman proposed using programmer annotated critical sections that should be considered for offloading to the large core to speed up execution and reduce the window for scalability bottlenecks to form by offloading the critical section. The results found in this work showed that accelerating scalability bottlenecks does help increase the scaling of the overall system. One of the main problems with the ACS system was the cost of migrating critical sections from the small cores to the large core. Another problem was determining the correct critical sections to offload and serializing on the large core doing the acceleration as it was a limited resource. A later proposal Suleman et al. [102] proposes a technique to push data to specialized cores to mitigate some of the problems seen in the ACS proposal. Both these proposals are methods to develop efficient techniques for using AMP systems. It should be noted these are primarily architecture techniques that look to leveraging the AMP in a fine grained fashion. They assume fast thread migration to get the maximum benefit from the AMP and prefer to limit involvement by the OS.

Dreslinski [52] tackles the problems with the ACS proposal involving moving critical sections that need accelerating to a remote core. Dreslinski proposes moving the powerful core to the critical section with fast voltage switching. This allows a boost in performance for a short period of time to reduce the runtime of the critical section and reduce the window for an scalability bottleneck to form in a similar fashion to ACS. The fast voltage boosting is enabled by further developments by Dreslinski in the area of near-threshold circuits [51] and architecture designs that allow for the fast voltage boosting. Still there are problems with determining which critical section being executed will form a bottleneck. But unlike ACS, with this proposal there is no longer a problem with moving the critical section and its data off to a remote core.

Interest in AMPs has also begun to be implemented by the industry. The main difference is the granularity of the industry implementations. Intel debuted their TurboBoost [47] technique in 2008. It was designed to allow their multi-core chips to automatically overclock a busy core when the remaining cores were idle. This was possible because the cores in most current Intel chips can run faster but they are clocked slower to meet worst case thermal budgets when all components in the chip are active. The main draw to the TurboBoost technique is to improve response time and overall user experience by improving

the performance of single threaded application for a short period. The individual core that is being boosted in Intel's technique can move around to improve performance. Marvell has also released an AMP called the Armada [62]. It uses multiple cores designed for difference performance and energy points. The Armada processor has two high performance cores to improve user experience and one low power, low performance core when performance requirements are less. This is closer to the commonly accepted design for an AMP. As industry and researchers move forward, it appears that AMPs are becoming more common. Research into techniques to leverage them are becoming more important to allow full utilization of the unique characteristics of AMPs.

In this thesis I will present techniques that build off of HTM contention management to determine the which transaction is the bottleneck to direct thread scheduling in an AMP. In particular I use the proposals by Dreslinski et al. to avoid the need to develop a fast thread migration facility.

2.7 Multi-Thread Fetch Policy

Related to the area of AMP thread scheduling is managing resource sharing within MT processors and giving the threads access to the proper resources to get optimal throughput. There has been appreciable work in MT processor fetch policies over the years. Most of the work in fetch policies has concentrated primarily in the area of SMT fetch policy. The original papers on SMT by Tullsen et al. [108, 107] concentrated first on describing the hardware necessary to enable SMT and then on fetch policies to optimize utilization. The most effective found in the work [107] was the ICOUNT policy. ICOUNT tracked the number of instructions from each hardware context in the pipeline and gave fetch priority to the thread with the least amount of instructions present. This is turn gave priority to the most efficient thread and generally provided maximum throughput. Work by Raasch and Reinhardt [89] looked at SMT fetch from the view of interactive applications and background tasks where absolute throughput was not the main indicator of acceptable performance. Latency was more important for their applications. Less efficient threads could be interacting with a user where ICOUNT would result in unacceptable performance by penalizing it, resulting in higher latencies being seen by the user. Raasch proposed priority schemes to be implemented in the processor to combat the latency problem with the previously proposed SMT fetch policies by giving a foreground thread elevated priority over all other threads to minimize interference from background threads. Raasch found that the priority scheme effectively reduced the latency penalty imposed on less efficient threads, but in turn decreased the overall throughput realized. Later work by Everman and Eeckhout [55] pro-

posed SMT fetch policies that tried to maximize the number of outstanding long latency load misses to increase overall performance by taking advantage of the high bandwidth memory subsystems of modern processors.

More recent work in the area of MT fetch policies was done by Lakshminarayana and Kim [75] in the context of Graphics Processing Units (GPUs). GPUs use large arrays of simple one-way in-order processing cores, many of the SMT fetch policies devised by Tullsen and others do not apply. Instead current GPUs use fine-grained MT with a Round-Robin (RR) fetch policy. Lakshminarayana and Kim looked at fetch many different fetch policies and found that a "Fair" policy was best for heavily threaded applications running on a GPU. The "Fair" policy kept a count of instructions that had executed on each core, and the fetch policy biased fetch to threads that had executed the least number of instructions. This turned out to be the best policy because of the nature of the programs running—highly independent threads executing until a barrier—and load balancing by tracking number of instructions executed gave appreciable speedup for these types of applications. As can be seen there has been multiple works in all the topics covered by this thesis with the overarching goal of increasing profitable use of parallel systems. In this thesis I will cover an MT fetch policy in Chapter 6 that shows unfair partitioning of resources among threads is best.

2.8 Summary

This chapter presented the background and related work in the areas pertinent to this thesis. The main theme running throughout this work is that managing CMPs running non-optimal parallel programs with scalability bottlenecks is difficult. The traditional methods required explicit programmer input to control this management and assumed the programs were scalable by default. In the past this was a valid assumption due to the small niche parallel programs used to fill. This is not an adequate solution for general purpose use of CMPs as required by today's programs. I propose that dynamic, at runtime management of CMPs is a solution to better leverage future CMPs for these new types of parallel programs. The following chapters will present solutions for managing scalability bottlenecks in general HTM systems, AMPs utilizing HTM, and MT cores with HTM support. In all of them I propose dynamic runtime management to enable better more scalable performance.

CHAPTER 3

Proactive Transaction Scheduling

In this chapter I present and evaluate the "Proactive Transaction Scheduling" technique for managing contention in HTMs. This chapter expands on the previously published work from [27]. I present the motivation behind the technique and a more detailed description of the implementation than from the previously published work. The evaluation section is also more detailed than the previously published work, offering more analysis and sensitivity studies. This chapter also provides clarifications to the previously published work on the operation of the algorithm.

3.1 Motivation

Contention management design is an important consideration when building an HTM as shown in Chapters 1 indicating contention is a real problem. The goal of an effective contention manager is to maximize the concurrency of the system by ensuring forward progress and preventing transactions from repeatedly aborting and restarting due to conflicting accesses. Contention is not a large problem in most cases at processor counts of less than or equal to 8 as shown in Tables 1.1 and 1.2 from Chapter 1. But as systems scale to higher processor counts (greater than or equal to 16), the problem of contention between transactions is exacerbated. A common method to handle contention is randomized backoff. It is an extremely simple and low-cost contention manager. While it works well at low processor counts, as the number of processors and threads increase, the effectiveness of randomized backoff rapidly decreases. Referring to Table 1.2, it can be seen that the conflict rates steadily increase for the STAMP benchmark suite running on an Eager Commit/Eager Conflict Detecting HTM, similar to LogTM [84] using randomized linear backoff for contention management. For all but the *Ssca2* benchmark, contention is a problem and limits scaling over sequential code. Of particular interest is the *Genome* and *Kmeans*

benchmarks. For these two, contention becomes a large problem at 16 processors and in both cases the performance scaling reverses and is worse than the performance at 8 processors. Likewise for the *Intruder* benchmark, at 8 processors its performance falls below that of the 4 processor configuration and at 16 processors this trend continues with performance dropping to 3x worse than serial performance. Illustrating the effects of contention on an Eager/Eager HTM is important because Eager/Eager HTMs require less hardware support than Lazy/Lazy HTMs and perform satisfactorily even though Lazy/Lazy HTMs perform better in general as seen in work by Shriraman et al. [99]and Tomic et al. [106]. Requiring less hardware is especially important for industry to adopt HTM. Industry will use the techniques that require the least amount of modification to current architectures to limit the extra validation effort. This is the most likely explanation behind choosing an Eager/Lazy HTM implementation—which requires even less hardware support than Eager/Eager—for the Sun Rock [48] and the AMD ASF proposal [44, 45] even though Eager/Lazy HTM experiences the most contention of all flavors of HTM as seen by Bobba et al. [36]. Finding more effective contention management schemes is very important as the STAMP benchmarks are indicative of future transactional applications where the transactions are coarsely placed more in line with a less than expert programmer.

3.1.1 Programmer Managed Contention Management

One method to reduce contention is to require the programmer (or compiler if available) to inspect the program and make changes that reduce the size of and minimize potential conflicting accesses inside a transaction. There are four common programming methods that can be used to reduce conflicts: Minimizing transaction size, selective cache line marking, open-nesting and early release. As covered in Chapter 2 this requires the programmer to inspect each transaction, including all function calls a transaction may contain and analyze it to determine what optimization can be used to shrink the transaction safely. As shown in Chapter 2, even for inserting into a linked list, these optimizations can be non-trivial and even different for each type of optimization. If it is required for the programmer to make such optimizations to get acceptable performance in transactional programs, then transactional programming is no easier than locks and will become just another tool to use in the appropriate situation. While having this support built into an HTM would be beneficial for expert programmers and optimizing compilers, more transparent solutions that offer acceptable performance should be pursued.

3.1.2 Reactive Contention Management and Theory

As covered in Chapter 2, automatic methods hidden to the programmer must be developed to attain acceptable performance in HTMs. One such method is contention managers. One of the simplest shown in Chapter 2 was reactive contention management. But as seen Tables 1.1 and 1.2 in Chapter 1, a reactive contention manager fails for the STAMP benchmarks.

To understand why randomized backoff contention managers may perform poorly, studies from the network domain that deal with similar problems of contention for a network link were investigated. In "*Performance Analysis of Exponential Backoff*" by Kwak et al. [74], the authors derive a mathematical model of an ethernet system to understand how it performs as the number of parallel transmitters contending for a shared ethernet wire is increased. As contention to transmit on the network increases, they found that the probability of successful transmission decreased to $\sim$35%. Qualitatively, this is what is seen with the STAMP benchmarks in Tables 1.1 and 1.2. As the number of parallel transactions in the system increases, the probability of successfully completing becomes steadily smaller. The final conclusion of Kwak's et al. is that randomized backoff is the best solution for a system that has no knowledge of what is trying to run on a contended resource even though it performs poorly under high contention. However, in the case of transactions the system does have knowledge of what is contending for shared resources. This opens the possibility for a better solutions to be designed. The next section will empirically show the existence of conflict patterns in benchmarks, which motivates the development of a predictor to manage contention and improve upon randomized backoff.

3.1.3 Conflict Locality

The STAMP benchmark suite is growing in popularity among researchers in the transactional memory field because of its long-term vision of how TM will likely be used in future applications. Works from Ramadan et al. [91], Bobba et al. [35], Bludell et al. [34], Dragojevic et al. [50] among others have turned to the STAMP suite to evaluate their systems. In early TM research, researchers used existing parallel programs from suites such as SPLASH2 [110], which were highly tuned parallel codes with very small transactions. But these are not very representative of future parallel programs because of this high optimization. In contrast, the STAMP suite uses large coarse grained transactions in relatively poorly tuned parallel code, which is considered more representative of future uses of TM, i.e. parallel programming for everybody else. This allows researchers to better predict and solve future problems TMs may face. In particular, the STAMP benchmarks show rising

amounts of contention as the number of processors is increased when using a randomized backoff technique.

In this chapter I propose using prediction of future conflicts between pairs of transactions to avoid contention by scheduling around these predicted conflicts using a software runtime. I base this proposal off the intuition that seeing conflicts in the past between transactions is an indicator that the conflicts will continue to happen in the future. The scheduling technique depends on the number of conflicts experienced by each transaction to be relatively small and non-random. A unique conflict is a conflict between two critical sections of separate threads. A non-random and small conflict set would mean the resultant conflict graph seen should not be very dense indicating that scheduling can be done to avoid pathological contention by disallowing pairs of transactions to run concurrently. If conflict graph is dense, it could indicate that scheduling would be useless, as there are too may have future conflicts, meaning the best case is to serialize. In this case, scheduling would be a high cost backoff algorithm because of the extra overhead scheduling requires to perform the scheduling decisions and therefore would be likely worse or equivalent to a simple backoff scheme. To evaluate whether scheduling by using past conflicts to predict future conflicts is feasible to a first degree, transactions were tagged in the program with a transaction ID (TxID) that was a concatenation of thread ID and a number assigned to the transaction in the code. The STAMP benchmarks were then run and all conflicting transaction pairs were recorded. Table 3.1 shows the resulting conflict graph for all tested STAMP benchmarks. It can be seen that the STAMP benchmarks show a wide range of conflict patterns, from very sparse conflicts as seen in the *Genome* and *Ssca2* benchmarks to very dense conflict patterns as seen in *Delaunay*. This implies that scheduling may be good only for a subset of benchmarks as it will work to prevent future conflicts by being more aggressive in serializing transactions over randomized backoff managers. A point to note is that Table 3.1 shows the cumulative conflict graph after a full execution of the benchmarks. It does not capture the dynamic conflict graph as transactions may go through periods of high contention and low contention where the conflict graph evolves. This behavior is demonstrated in the "Adaptive Transaction Scheduling" paper by Yoo and Lee [113], and this behavior makes the idea of scheduling more promising than the measured conflict graphs indicate. In the next section I cover the implementation details required to build a *Proactive* transaction scheduler that uses the dynamic conflict graph to make predictions about future conflicts and therefore improve overall performance.

Benchmark	Tx	Conflict Graph	Benchmark	Tx	Conflict Graph
Delaunay [72]	0:	0 1 2	Intruder	0:	0
	1:	0 1 2 3		1:	1 2
	2:	0 1 2 3		2:	1 2
	3:	1 2 3			
Genome	0:	0	Ssca2	0:	0
	1:			1:	
	2:	2 3		2:	2
	3:	2			
	4:	4			
Kmeans	0:	0	Labyrinth	0:	0
	1:	1 2		1:	1 2
	2:	1		2:	1 2
Vacation	0:	0	Yada	0:	0 2 4
				1:	2
				2:	0 1 2 3 4 5
				3:	2
				4:	0 2 4 5
				5:	2 4 5

Table 3.1: Conflict Group Set for the transactions in the STAMP Benchmarks.

3.1.4 Proactive Contention Management

As described in Section 3.1.2, the effectiveness of randomized backoff in managing contention degrades rapidly as the number of cores increases. But, in a TM system we can determine when there may be contention for resources, and therefore provide smarter contention management than backoff. I propose scheduling transactions in place of randomized backoff to manage contention in the previous section by looking at generated conflict graphs of the STAMP applications. This idea leverages two key observations: most applications that benefit from parallel programming have large problem set sizes and are mostly throughput oriented, so there should always be another thread ready from the current program to do other independent work that can be swapped in to allow better forward progress. Another observation that we validate later is that transactions conflict in predictable patterns and that there is extra parallelism to exploit by swapping threads instead of simply stalling processors and wasting resources. These observations are used to implement a technique called "Proactive Transaction Scheduling" (PTS) that can extract better performance dynamically at runtime from an overcommitted CMP system and using the dynamic conflict graph to make scheduling predictions by predicting conflicts between transactions. An overcommitted system is key to the PTS technique. By having more threads than processors, another thread can be swapped in to potentially do other inde-

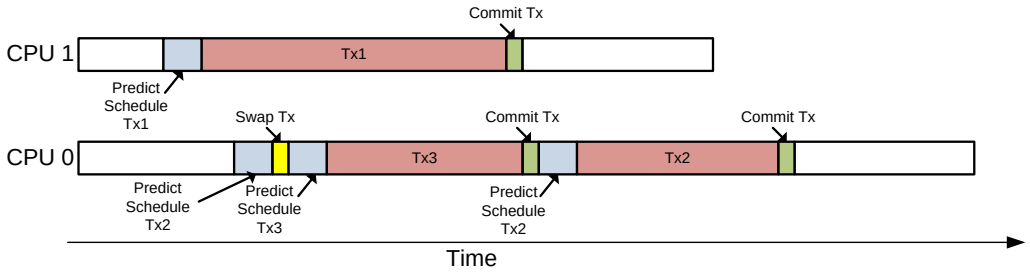


Figure 3.1: Proactive Transaction Scheduling contention management example operation.

pendent work. This in effect hides the latency of serializing transactions, which is a key contribution of this thesis. Many researchers still assume the number threads will equal the number of processors which is wasteful when serialization must take place. Figure 3.1 shows a simple example of how PTS will manage contention and improve performance by adding some overhead at the start and end of each transaction. The example shows a two processor system trying to execute three transactions. By using the past conflict history, a scheduler can predict that it is likely that Tx2 and Tx1 will conflict. The scheduler can suspend the thread trying to execute Tx2, and execute Tx3 (which runs within a different thread) in its place. This improves performance by performing other useful work in place of a transaction that is likely to conflict with another concurrently executing transaction. Another way to think of PTS is that it is dynamically creating blocking synchronization (locks) or optimistic synchronization (transactions) when appropriate. Unlike backoff techniques that stall transactions to avoid conflicts, this technique can bring in new work to maximize useful computation instead of spending it stalling. Conflict history is also continuously updated so blocking synchronization decisions do not immediately revert back to optimistic synchronization that would allow conflicts to reoccur.

3.2 Implementation

The PTS contention manager is implemented as a user-space software runtime layer built on top of an Eager/Eager Transactional Memory system implemented in the M5 simulator [26] as shown in Figure 3.2. The HTM works in conjunction with the software runtime to implement PTS. The scheduler is a *fully distributed* algorithm that each processor runs in parallel whenever a transaction wishes to begin execution. First, each processor looks at a snapshot of what transactions are currently executing on the system and gathers information about conflicts from the global transaction conflict graph stored in memory. Each processor then locally decides its probability of generating a conflict from information derived by these lookups and decides the appropriate course of action: swap in a new

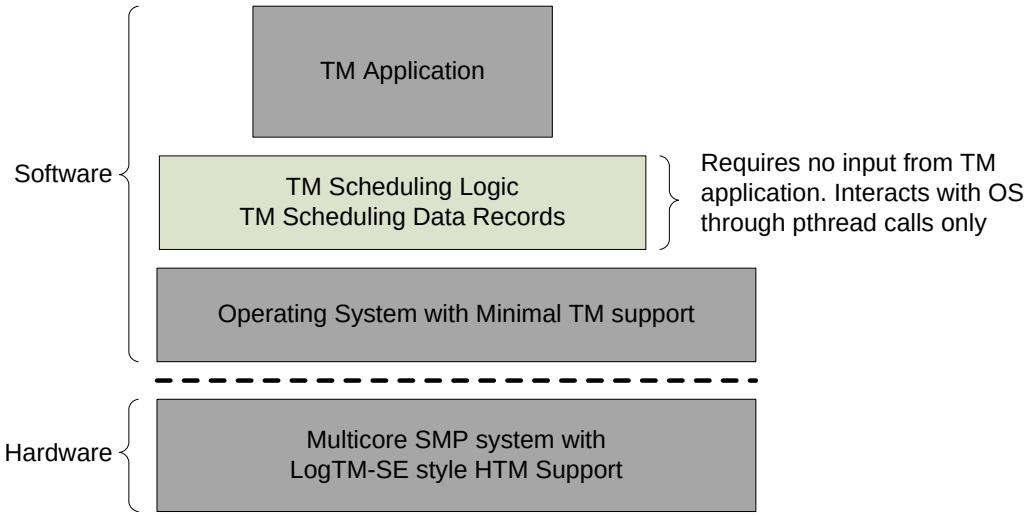


Figure 3.2: Hardware/Software stack of our proposed system

thread, stall briefly or begin execution. None of the processors running the scheduling algorithm explicitly communicate their intentions to each other, nor is the system snapshot necessarily consistent when viewed by multiple processors because it is updated without using any form of synchronization. While enabling communication or globally consistent snapshots could be beneficial, the synchronization necessary to provide such facilities would be overly costly. The following subsections describe the pieces that constitute this system and how they fit and work together.

3.2.1 Hardware Additions

PTS is built assuming an Eager/Eager transactional memory model developed inside the M5 Full System simulator. The design closely resembles the original LogTM [84] and also uses signatures like LogTM-SE [111] and details of the base implementation can be found in a workshop paper [30]. To enable logging the necessary data for the software runtime additional functionality is added to the CPU, cache controllers and coherent interconnect.

The CPU and interconnect modifications are shown in Figure 3.3 and consist of additional registers and an out-of-band data channel which are bolded in Figure 3.3. The next set of registers are specific to the PTS contention manger, though they could be used by other contention managers. The *TxID* register holds the ID of the currently running transaction. The *TxSize* register is updated on transaction commit with the total size of the RWSet. The final set of registers hold RWSet summaries in the form of Bloom filters [32] that can be accessed by the software scheduling runtime. To be effective the Bloom filters have to be large and use efficient hashing functions like the H_3 hash functions [39]. The

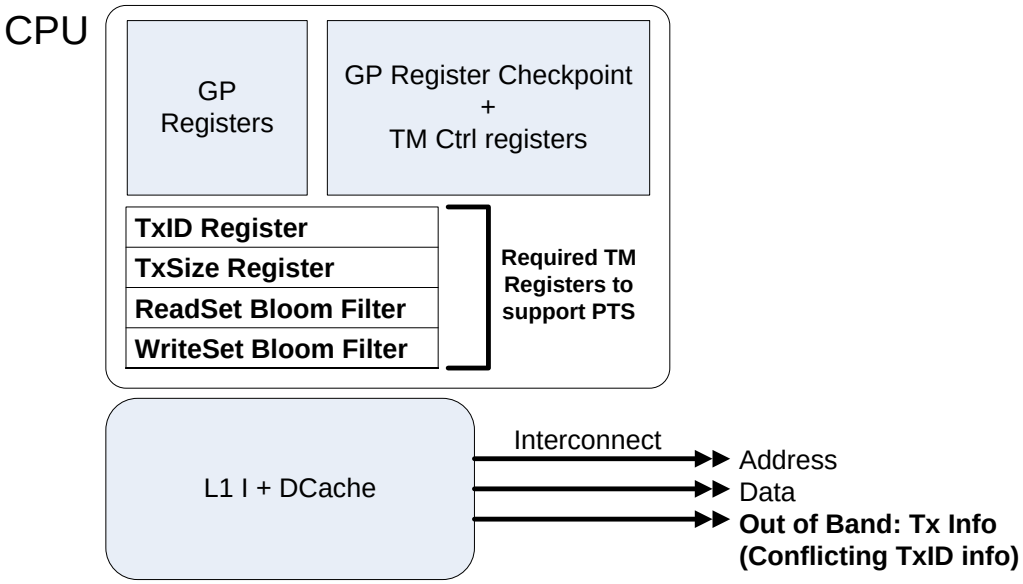


Figure 3.3: Additional registers and interconnect extensions to support proactive scheduling. New additions are bolded.

experiments in this chapter show that sizes between 512bits and 8192bits are effective using only a single hash bit. Work by Cao Minh et al. [38] also found larger Bloom filters were better. This thesis assumes that Bloom filters of this size can be built using similar techniques developed by Yen et al. [112] and Sanchez et al. [95].

The second modification is to the coherent interconnect and the cache controllers. When the cache has to notify a remote transaction it must abort due to a conflict, the cache controller first gets the value of the *TxID* register from the CPU. Then it sends a response back to the remote processor containing this *TxID* over the interconnect as an out-of-band data response. When the remote cache receives this response, it passes the conflicting *TxID* value back to the remote processor to use to update the conflict information. This conflicting *TxID* is stored in one of the general purpose registers for direct use by the *abort routine* which is immediately vectored to by the CPU when a conflict occurs.

3.2.2 Proactive Scheduling Runtime

The majority of PTS is implemented as a software runtime, as many of the operations covered in this section would require large amounts of hardware to implement. The software runtime is implemented as a user-space thread scheduler. This design was chosen so the cost of calling into the scheduler at the start of every transaction is minimal. We use the `pthread_yeild()` function from the pthreads library provided in Linux to suspend threads and force the operating system scheduler to execute threads in an order determined by the

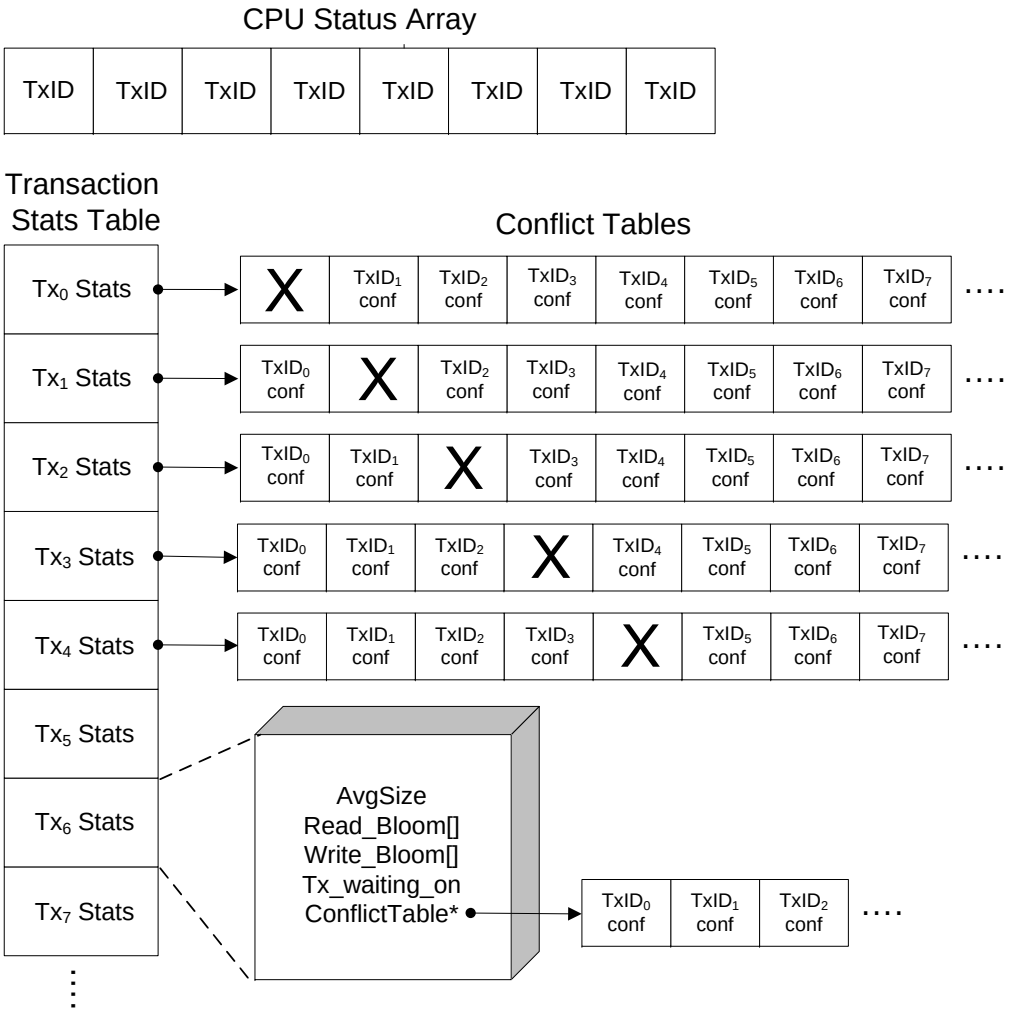


Figure 3.4: Data structure representation for an example 8 CPU system.

proactive scheduler. The runtime uses three global data-structures and three main function routines to implement the distributed PTS algorithm.

3.2.2.1 Data Structures

There are three global data structures used by PTS. These are the *CPU Status Array*, *Transaction Stats Table*, and the *Conflicts Table*. These data-structures provide the snapshot of the current transactions executing in the system and the conflict history in the form of a graph along with other statistics useful for doing scheduling operations in PTS. An example of the required data structures for an eight processor system is shown in Figure 3.4.

CPU Status Array: The *CPU Status* array is a globally accessible array that is sized to the number of processors present in the system. As seen in Figure 3.4, for an eight processor system the array is of size eight. The array contains the information of the transaction ID

that is currently running on each processor by storing its TxID in that processor's corresponding entry.

Transaction Stats Table and Conflict Table: The next two data structures hold information about each individual transaction and also maintain information about past conflicts which is represented as a dependency graph using a full matrix representation. The *Transaction Stats Table* is a global table shared by all threads, and each entry in the *Transaction Stats Table* has its own *Conflict Table* which is a row of the conflict graph matrix. Each *Conflict Table* entry holds a saturating counter *conf* that represents the confidence of a conflict occurring in the future between a pair of transactions. A *Transaction Stats Table* entry stores information such as the *AvgSize* variable, used to indicate the average historical runtime of this transaction represented using the overall number of cache lines touched during execution. The table entry also holds a Bloom filter representation of the most current successful commit of the transaction's RWSet. The purpose of these filters will be covered in the next section. The *Tx_waiting_on* variable tracks the most recent remote transaction the local transaction has serialized behind. Its use will also be covered in more detail later.

In our design, the *CPU Status Array* is implemented as a fixed sized array which is allocated at program start since the maximum number of processors is known. The *Transaction Stats Table* is also allocated at program startup along with the *Conflict Tables* because the number of unique transactions that can exist in the system is set at compile time and the maximum number of TxIDs is passed to the scheduling runtime library at program start. This requires a rather large memory footprint, on the order of $O(N^2)$, but offers an $O(1)$ time to access any part of the table or conflict graph matrix, reducing the overhead for each invocation of the scheduler. In our experiments our tables grew to a maximum of 50MB for benchmarks with a large number of TxIDs. This means that PTS and its accompanying data-structures could only be implemented in software. However, more space efficient representations can be constructed as data-structures of this size could lead to bad cache performance due to its shared nature and size.

3.2.2.2 Scheduler Algorithm Implementation

The scheduler has three main routines that form the main portion of the distributed algorithm. These functions work to schedule transactions, update the conflict graph, update the current snapshot of executing transactions, and update transaction statistics. The main functions are `scheduleTx()`, `txConflict()` and `commitTx()` and are described below. The scheduler is a parallel program in its own right, any of these three routines can be executed by any or all of the processors concurrently.

scheduleTx(): The `scheduleTx()` function is called before the start of any transaction and

Example 3.1 Schedule Transaction Pseudo Code for the PTS algorithm

```
1  void scheduleTx(int TxID)
2  {
3   start_schedule_loop:
4   for(int i=0;i<sizeof(cpuStatusArray);i++)
5   {
6     if(i!=ourCPU)
7     {
8       remoteTxID=cpuStatusArray[i];
9       if(confProb(TxID, remoteTxID) > confThreshold)
10      {
11        logTxWaitingOnVar(TxID, remoteTxID);
12        if(txSizeThreshold >= checkSize(remoteTxID))
13        {
14          doSmallRandomBackoff();
15          break;
16        }
17        else
18        {
19          pthread_yield();
20          goto start_schedule_loop;
21        }
22      }
23    }
24  }
25  cpuStatusArray[ourCPU]=TxID;
26 }
```

is shown in Example 3.1. It is rather simple and works by scanning the *CPU Status Array* for TxIDs that could potentially conflict with the TxID wanting to execute. A conflict is predicted by indexing into the conflict graph matrix, using `TxID` as the row index and the `remoteTxID` as the column index to get a confidence value. If the confidence value is below the `confThreshold` (in the presented experiments the `confThreshold` is set to 5 and the confidence ranged from 0 to 10), the algorithm continues scanning, otherwise it decides how to serialize the transaction. This algorithm greedily picks the first transaction executing predicted to conflict in an effort to reduce overhead instead of picking the maximum confidence value. If a conflict is predicted the function then decides if the predicted conflicting remote transaction is "small" or "large" by indexing the *Transaction Stats* table. If the transaction is large, which is done by looking at the *AvgSize*, then the scheduling function calls `pthread_yield()` to force the currently running thread to the back of its run queue in the Operating System (OS). The OS will then swap in a new thread that will try to

execute its transactions. Large transactions are set as greater than 10 cache lines in size in the experiments presented later in this chapter. Upon return from `pthread_yield()`, the function jumps to the beginning of the scheduling function and re-executes the scan of the *CPU Status Array*. If the remote transaction is predicted to be small then a simple random backoff is initiated to stall the current local transaction for a short while before letting it execute without re-executing the scan of the *CPU Status Array*. This is done because calling `pthread_yield()` is expensive and unnecessary when predicting it necessary to serialize behind short remote transactions. This feature also makes the scheduler more optimistic than if it stalled and then re-executed the scan of the *CPU Status Array*. When the local transaction predicts it does not conflict with any other running remote transaction in the system, it sets its TxID in the *CPU Status Array* and executes.

It is important to note that all the processors could begin running this routine at the same time because it is a *distributed* algorithm. Because there is no explicit synchronization or communication among processors i.e. locks or message passing, the scanning of the *CPU Status Array* may yield stale information as the processors running the routines are in a benign data race to complete their scans and update their respective entries in the *CPU Status Array*. This may lead to unintended conflicts because each processor may schedule conflicting transactions without realizing it due to inconsistent views of the *CPU Status Array* caused by these data races. Still, this is desirable over inserting synchronization to only allow one writer at a time to the *CPU Status Array* because the cost of such synchronization is high and deadlock is not an issue. On the other hand, starvation could happen if a pathological condition happened where all processors predicted no conflict, and kept conflicting because they did not see updates to the *CPU Status Array* because of this race condition. Throughout the experimentation no issues with starvation were experienced.

Example 3.2 Conflict Handling Pseudo Code for the PTS algorithm

```
1 void txConflict(int TxID,int confTxID)
2 {
3   cpu_status_array[ourCPU] = NO_TX;
4   incConflictProb(TxID, confTxID);
5   incConflictProb(confTxID, TxID);
6   if (txSizeThreshold >= checkSize(confTxID))
7     doSmallRandomBackoff();
8 }
```

txConflict(): When a transaction conflicts with another transaction, the transaction is first rolled back. Then the `txConflict()` routine as shown in Example 3.2 is called to update the conflict graph matrix of the transaction pair that conflicted. The routine accesses each

transaction's *Conflict Table* and looks in the entries for the respective TxIDs. The conflicting TxID is obtained as discussed in Section 3.2.1, the processor stores the ID in a general purpose register that can be accessed by these routines easily. If the conflict has never been seen before, the confidence is initialized to a default value (in the presented case the default value is set to 5, and the counter saturated at 10), otherwise the confidence counter is incremented by one.

Example 3.3 Commit Transaction Pseudo Code for the PTS algorithm

```
1  void commitTx(int TxID)
2  {
3    updateBloom(TxID);
4    updateAvgSize(TxID);
5    cpu_status_array[ourCPU] = NO_TX;
6    int TxWaitingOn = checkWasSerialized(TxID);
7    if (TxWaitingOn != NO_TX)
8    {
9      if (intersectBlooms(TxID, TxWaitingOn))
10       incConflictProb(TxID, TxWaitingOn);
11     else
12       decConflictProb(TxID, TxWaitingOn);
13   }
14 }
```

commitTx(): On commit, transactions call the `commitTx()` function which is shown in Example 3.3. In this function statistics such as average size and the current bloom filter are updated for the transaction in its entry in the *Transaction Stats Table*. First the thread erases its entry in the *CPU Status Array*. Next the thread saves its current Bloom filters. Finally the committed transaction checks to see if it had been waiting on another transaction sometime in the past by checking the *TxWaitingOn* variable in its *Transaction Stats Table* entry. If it was serialized behind another transaction, it obtains the most current Bloom filters from the table entry pointed to by *TxWaitingOn* to compare against its own. If the two Bloom filters intersect, the thread increments the confidence of conflict with that transaction by one, otherwise it decrements the confidence by one. This last part using Bloom filters to update confidence is vitally important as it is the method by which transactions can be identified that have diverged and can predict that they will no longer conflict, thereby allowing pairs of transactions to resume using optimistic synchronization instead of pessimistic synchronization. This allows the conflict graph witnessed during execution to evolve.

The `commitTx()` is an interesting function and exposes some properties of Bloom filters

that have to be discussed as the operation is fundamentally different than those traditionally used with Bloom filters. Bloom filters are primarily used to test set inclusion, meaning that an element is hashed to a value that is then intersected with the Bloom filter that represents the entire set. The hash used usually sets multiple bits to reduce the chance of false positives when testing set inclusion. In fact, the basic probability theory behind Bloom filters states that setting more hash bits is better when testing set inclusion, this is also found by Sanchez et al.'s [95] studies. On the other hand, the operation being conducted by the `commitTx()` function is a set *intersection* function, two fully populated Bloom filters are being bitwise ANDed together instead of a single element hash bitwise ANDed to a Bloom filter. In this case, setting more bits per entry into the filter is counter productive. As stated in Section 3.2.1, the Bloom filters use only use 1 hash bit per entry to reduce false positives for set intersection. This reduction in false positives is caused by the sparseness of a large Bloom filter using only 1 bit per entry in the Bloom filter representation of a set. If more bits were used per entry in the Bloom filter the likelihood of an intersection operation returning common bits is more probable. Because the Bloom filter retains no information as to which element each bit corresponds to, it cannot be determined if the returned intersected bits are an actual entry in common between the two filters or a false positive composed of bits from multiple entries. This can be explained mathematically in the following equations which are derived originally in [32].

Equation 3.1 shows the approximate probability that any single bit in a Bloom filter is still 0 after inserting n elements using k hash functions for a Bloom filter of size m.

$$P_0 \cong e^{\frac{-kn_1}{m}} \tag{3.1}$$

This can be used to derive the probability of at least one common bit being set in two Bloom filters that may or may not contain a common element. The probability of this is shown in Equation 3.2.

$$P_{1bitintersects} \cong 1 - (e^{\frac{-kn_1}{m}})^{kn_2} \tag{3.2}$$

The above equation is also the probability of the Bloom filter intersection operation used in the `commitTx()` returning true for a Bloom filter using just one hash function $k = 1$. This equation can be generalized for any number of k by accounting for the situations where the intersection operation returns less than k bits in common, signifying no intersection. This is presented in Equation 3.3.

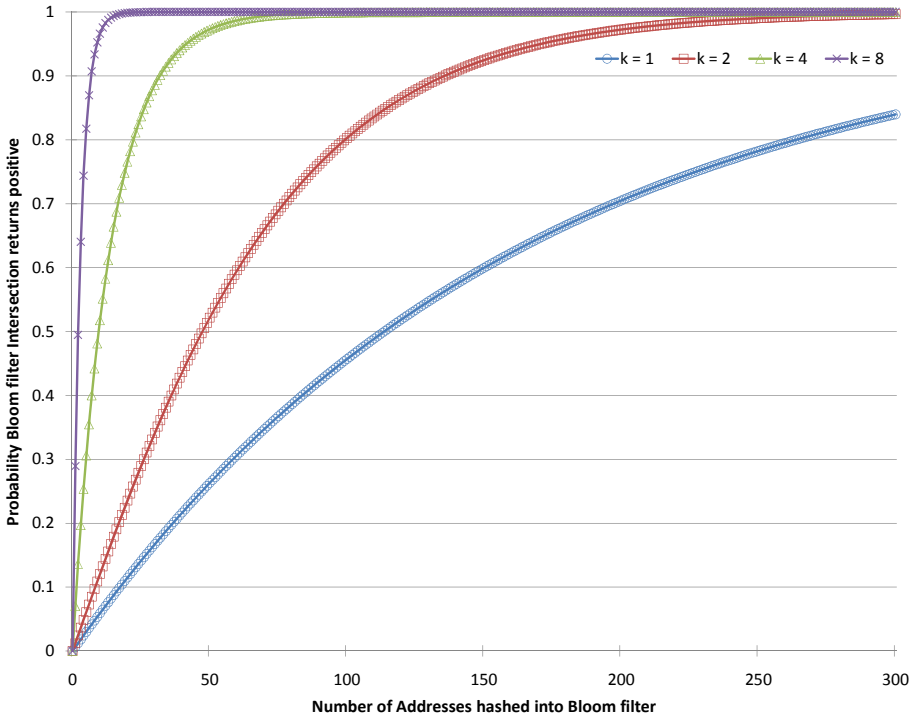


Figure 3.5: Probability of an intersection returning elements in common for a 8192bit Bloom filter being intersected with another 8192bit Bloom filter that has 50 addresses hashed into it with number of hashes (k) set to 1, 2, 4, and 8.

$$P_{intersection} \cong 1 - (e^{\frac{-kn_1}{m}})^{kn_2} - \sum_{j=1}^{k-1} \binom{kn_2}{j} (1 - e^{\frac{-kn_1}{m}})^j (e^{\frac{-kn_1}{m}})^{(kn_2 - j)} \tag{3.3}$$

Using Equation 3.3, Figure 3.5 is generated for intersecting two 8192bit large Bloom filters with one having only $n = 50$ addresses hashed, and the other with a varying n entries hashed into the filter. As shown in the figure, $k = 1$ is the best number of hash functions when performing an intersection operation between Bloom filters.

3.2.3 Proactive Scheduling Runtime Optimizations

During the development and testing of the PTS contention manager, various optimizations were made to further enhance performance. Two specific optimizations were found. One optimizes the behavior of small transactions (transactions touching less than 10 cache lines in their RWSet), and the other makes predictions more optimistic for specific transactions in code that display random behavior.

3.2.3.1 Small Transaction Optimization

As described in the previous sections, the PTS algorithm is written completely in software, and therefore adds overhead to the start and end of every transaction. If the transaction is small, on the order of less than 10 cache lines, then it is advantageous to minimize overhead as much as possible. Small transactions are unlikely to cause long lasting conflicts and should be able to be predicted more optimistically. Small transactions short-cut the prediction mechanisms using a contention pressure variable in a similar fashion to the ATS technique from Yoo et al. [113]. The contention pressure variable is stored in each transaction's entry in the *Transaction Stats Table*. When a conflict is detected, then the contention pressure value for the transaction is incremented. When a small transaction commits, it decrements the pressure value which in effect is predicting that conflicts are becoming less likely. On transaction begin, the contention pressure value is checked for small transactions and if the pressure value is below a threshold, the transaction is allowed to immediately begin execution omitting the scan of the *CPU Status Array*. If the contention pressure value is high, the transaction performs the normal scheduling operations as covered in the previous sections.

3.2.3.2 Splitting Transaction IDs for Hard to Predict Large Transactions

In some benchmarks, certain transactions conflict in random patterns that are not as predictable due to the data-structures that they perform operations on. These benchmarks are the *Delaunay*, *Vacation* and *Yada* benchmarks. One of the common factors these benchmarks have is a dense conflict graph as seen in Table 3.1, as well as working on large data structures that exhibit random transient conflict patterns that should be predicted optimistically. The *Delaunay* benchmark works on a graph data-structure representing a large 2D mesh of triangles that is refined by creating new triangles. Each iteration of the *Delaunay* algorithm works in a random location of the data-structure. Even though conflicts are common during this benchmark they are primarily transient. For the *Vacation* benchmark, the application builds a large Red-Black tree during execution. As the tree grows, operations on it become more disjoint and less likely to conflict and should be treated more optimistically. The *Yada* benchmark is very similar to the *Delaunay* benchmark. It also works on a large graph data-structure representing a 2D mesh of triangles. Like the *Delaunay* benchmark, operations in *Yada* experience many conflicts but they are transient in nature.

To counteract pessimistic over-serialization that can happen in PTS for transient conflict patterns, I found that splitting transaction IDs can be used to make predictions more optimistic. Splitting transaction IDs consisted of mapping a single transaction to more entries in

the *Transaction Stats Table* and *Conflict Table*. As the program executes transactions using split IDs, PTS rotates through the extra entries using different prediction data for subsequent predictions. Doing so effectively causes resets of the confidence values periodically to allow for more optimistic predictions when dealing with transient conflict patterns. I show in section 3.3.4.2 that this technique works well for these particular benchmarks. The main problem with this technique is it required familiarization with the algorithms studied to determine that this optimization was applicable. Therefore it is not a general technique. Chapter 4 formalizes methods to identify and take advantage of this type of transaction behavior automatically. Originally the idea behind splitting transaction IDs was more application specific. It involved using program specific data to construct transaction IDs that would implicitly contain conflict information. An example would be the location in the 2D mesh to be modified by a transaction in the *Delaunay* application. Transaction IDs that were sufficiently far apart numerically would imply a conflict was not likely. This idea was very similar to the Transaction Executor idea proposed by Bai et al. [22]. Similarly to Bai's work, this method is not portable even within the same applications if the input data changes.

3.2.4 Hybrid Proactive Scheduling

A lightweight *hybrid* Backoff/PTS scheme was also developed to account for low contention cases where the PTS algorithm could be too expensive. This is shown in Section 3.3 with the *Ssca2* benchmark. The hybrid scheme starts using backoff as its contention manager. During execution it keeps track of the global conflict rate by tracking two global counters that are incremented atomically: number of transactions began and number of conflicted transactions. If the conflict rate reaches a preset threshold (5% in the presented tests), it switches to using PTS instead of backoff for the remainder of benchmark execution. This hybrid implementation contains no method to switch between backoff and PTS dynamically, but in Chapter 4 I cover an algorithm that is capable of doing so.

3.3 Evaluation

3.3.1 Simulation Environment and Benchmarks

To evaluate the PTS contention management technique I used the M5 Full System simulator modified to implement a custom version of LogTM. PTS is evaluated using the STAMP [37] suite modified to operate with M5's version of transactional memory. The STAMP benchmarks, descriptions and input parameters are presented in Table 3.2. All

Benchmark	Version	Description and Parameters
Delaunay [72]	v0.9.2	Refines a 2D mesh of triangles using Delaunay refinement. Input "-i inputs/large.2 -m30 -t64"
Genome	v0.9.2	Genome sequencing benchmark. Input "-g4096 -s32 -n524288 -t64"
Kmeans	v0.9.2	Kmeans clustering algorithms. Input "-m20 -n20 -t0.05 -i inputs/random50000_12 -p64"
Vacation	0.9.2	Simulates a multi-user database, modeled as a Red-Black tree. Input "-n8 -q10 -u80 -r65536 -t131072 -c64"
Intruder	0.9.10	Signature-based network intrusion detection benchmark that captures and reassembles packet streams for scanning. Input "-a10 -l32 -n8192 -s1 -t64"
Ssca2	0.9.10	An efficient graph construction algorithm using adjacency arrays and auxiliary arrays. Input "-s15 -i1.0 -u1.0 -l3 -p3 -t64"
Labyrinth	0.9.10	A transactional version Lee's routing algorithm[78] through a maze. Input "-i inputs/random-x96-y96-z3-n128.txt -t64"
Yada	0.9.10	'Yet Another Delaunay Algorithm' for refinement of a 2D mesh of triangles, uses a different algorithm from the other Delaunay algorithm implementation. Input "-i inputs/large.2 -m30 -t64"

*The Bayes benchmark is not evaluated because of its non-deterministic finishing conditions as noted in [37], which makes direct comparisons between contention managers inconclusive

Table 3.2: STAMP Benchmark descriptions, version used, and input parameters.

STAMP benchmarks except the *Bayes* benchmark are tested. *Bayes* is not tested because it has non-deterministic finishing conditions. The input sets used are not standard but are between the recommended simulation input sets and the input sets recommended for native hardware execution. Also to note is that all the simulation input parameters specify an over-committed system where there are more threads than processors. This is required by the PTS technique as specified in the motivation and implementation sections. These benchmarks stress the TM system, especially the contention manager as they can suffer high contention. Statistics are collected only during the parallel phase of each benchmark. For the *Labyrinth* benchmark, the code is modified to perform the grid copy outside of the transaction as has been done by others. This allows some parallel scaling as unmodified it operates serially.

The simulation parameters are presented in Table 3.3. The system used to evaluate the PTS proposal assumes an aggressive 16-core CMP system with a large amount of cache. As transistor budgets continue to grow, the proper balance between cores and cache space is still an open question. As seen by the contention experienced in the STAMP benchmarks, a large number of cores may not be the optimal solution as many of the applications

Feature	Description
Processors	16 one IPC Alpha cores @ 2GHz
L1 Caches	64kB, 1 cycle latency, 2-way associative, 64-byte line size
L2 Cache	32MB, 32 cycle latency, 16-way associative, 64-byte line size
Interconnect	Shared bus at 2GHz
Main Memory	2048MB, 100 cycles latency
Linux Kernel	Modified v2.6.18
Contention Managers	Randomized Linear Backoff, PTS, PTS-Backoff Hybrid
Signature Size	512bit-8192bit for PTS commit routines, perfect signature used for conflict detection.

Table 3.3: M5 Simulation Parameters.

have problems scaling to a modest system size of 16 cores. In this light, dedicating more die area to cache would be more beneficial. This trade-off is one of the reasons the system studied in M5 is designed the way it is. In this Chapter, I test a Randomized Linear Backoff contention manager against the proposed PTS and PTS-Backoff Hybrid contention management techniques.

Parallel programs can also show noticeable runtime variation due to interleavings of synchronization operations that cause different executions paths to be followed as shown by Alameldeen and Wood [16]. Therefore, the simulations are run multiple times using small random perturbations to the memory system to get a stable runtime average.

As seen in the previous section, there are many parameters that could be potentially investigated for sensitivity in the PTS technique. Because of the size of the search space, multiple parameters remain fixed while others are varied. Here I list out the parameters and specify which are fixed, and which will be studied in more depth by performing sensitivity tests: 1) Minimum, Maximum confidence values and conflict threshold - fixed to 0, 10 and 5 respectively. 2) Confidence increment/decrement value - fixed to increment/decrement by 1 when confidence is modified. 3) Small transaction threshold - fixed to 10 cache lines (approximately double the size of queue/dequeue operations in STAMP benchmark suite). 4) Bloom filter size - variable between 512bit-8192bit, uses k=1 H3 [39] hash function.

3.3.2 Performance Analysis

This section presents a performance analysis of PTS and the PTS-Backoff Hybrid contention managers as compared to a reactive randomized linear backoff contention manager for an Eager/Eager LogTM like HTM. The performance results are presented in Figures 3.6, 3.7 and Table 3.4. Figure 3.6 presents the speedup seen for a 16 core system over a 1 core system running the same benchmark. Figure 3.7 presents the percent difference

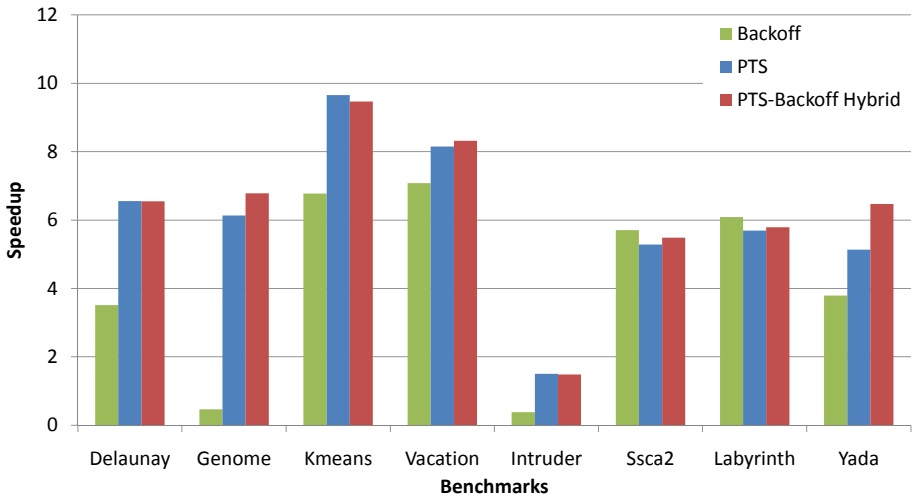


Figure 3.6: Overall best attainable performance of PTS and PTS-Backoff Hybrid compared to Backoff for a 16 processor system.

	Backoff	PTS	PTS-Backoff Hybrid
Delaunay	73.4%	26.6%	26.7%
Genome	49.7%	1.8%	1.7%
Kmeans	20.6%	3.7%	3.4%
Vacation	10.3%	8.5%	8.0%
Intruder	70.1%	7.5%	9.9%
Ssca2	<0.1%	<0.1%	<0.1%
Labyrinth	18.2%	21.5%	29.2%
Yada	54.6%	27.8%	21.8%

Table 3.4: Contention experienced for each contention management technique: Backoff, PTS, and PTS-Backoff Hybrid for a 16 processor system.

seen between Backoff and the PTS contention managers. Table 3.4 shows the percentage of conflicts seen for each contention management scheme. All the results in this section are presented using the best performing combination of Bloom filter size (512bit - 8192bit) and applied optimizations (small transaction short circuiting from Section 3.2.3.1 and split transaction ID optimizations from Section 3.2.3.2). Later sections will present sensitivity studies to provide a full picture as to the effects of these optimization and parameter choices.

The **Randomized Linear Backoff** contention manager performs the worst out of all the contention managers on average as seen in Figures 3.6 and 3.7. In the benchmarks *Genome* and *Intruder* the performance at 16 processors has become worse than serial. The

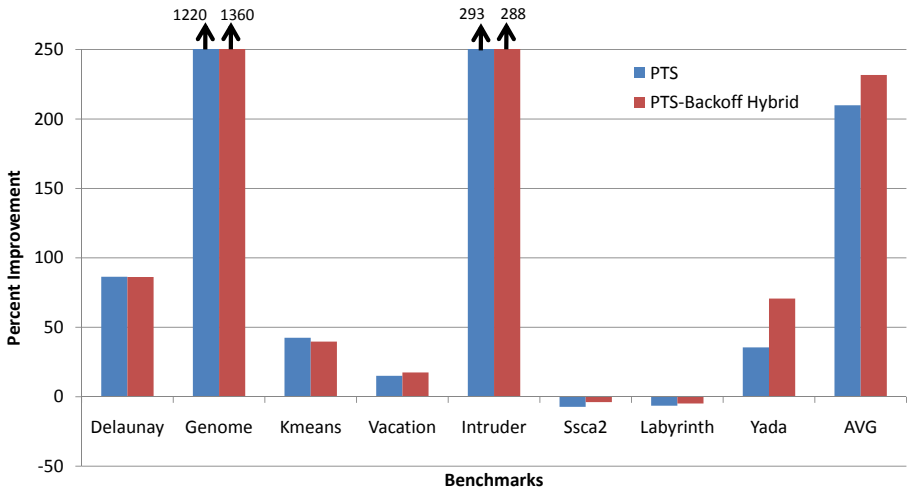


Figure 3.7: Percent difference of PTS and PTS-Backoff Hybrid over Backoff for a 16 processor system.

worse than serial performance highly correlates to large amounts of contention seen for the Backoff contention in Table 3.4. The other benchmarks that experience high contention such as the *Delaunay* and *Yada* benchmarks see very little scaling but the performance is not pathologically bad by dropping below serial performance. Backoff is still a useful contention management policy for very low contention benchmarks such as the *Ssca2* benchmark which has very small transactions. For this case Backoff is the best policy as it incurs almost no overhead except in the rare case of a conflict. *Ssca2* shows poor scaling not due to contention but instead due to poor memory layout and caching characteristics. L1 data cache misses exceeded 10% in the case of *Ssca2*. This thesis does not address such issues but numerous researchers are actively looking into efficient memory hierarchy management and program layout to solve problems such as what is seen in the *Ssca2* benchmark. The *Labyrinth* benchmark also performs best using the Backoff policy. This is due to *Labyrinth* having very random conflicts that are hard for PTS to predict properly.

The **PTS** contention manager performs very well compared to Backoff, attaining an arithmetic average performance improvement of 209% over Backoff as seen in Figure 3.7. The performance gains seen are highly correlated to the large reduction in contention as shown in Table 3.4. PTS is able to reduce this contention by an order of magnitude for benchmarks like *Intruder* that see very high contention when using a reactive backoff based contention manager. In general PTS allows benchmarks that appear to have limited scalability such as the *Delaunay* benchmark to almost double their performance to better use a 16 processor CMP. PTS also shows that using scheduling and predicting conflicts prevents pathological contention causing less than serial performance as in the *Genome* and

Intruder benchmarks. By eliminating the pathological contention case in *Genome*, this particular benchmark shows a decent amount of scalability. In the case of *Intruder*, PTS gets some performance, but the benchmark appears to have limited scalability in general, but PTS is able to deal with this and schedule accordingly to prevent worse than serial performance. One of PTS's main drawbacks is the overhead it incurs on every transaction. In the case of benchmarks like *Ssca2* which see limited amounts of contention this overhead is unnecessary, and therefore performance is less than low overhead techniques like Backoff. In the case of *Labyrinth*, PTS has a hard time making good decisions due to the benchmark characteristics and the small number of transactions it runs, limiting the amount of time the system gets to learn conflict patterns. In these cases where PTS loses, the amount of performance lost is small.

The **PTS-Backoff Hybrid** contention manager attempts to solve the one main problem PTS has and that is overhead on transaction begin by turning off PTS unless a high conflict rate is detected, for these experiments a high contention rate was set as 5% as seen by all transactions. This leads to an overall average improvement over backoff of 231%. For the *Ssca2* benchmark, this gains back some performance as seen in Figures 3.6 and 3.7. The slight performance loss is due to PTS-Backoff Hybrid still having to do a small number of operations every transaction begin to track a global contention rate while Backoff does no operations on transaction begin. In general PTS-Backoff Hybrid performs similar to PTS for the rest of the benchmarks due to their contention rates being high from the beginning, so PTS is on for the majority of execution time in the other benchmarks. In the *Yada* benchmark, PTS-Backoff Hybrid sees a sizeable performance gain. Examination of where time is spent between PTS and PTS-Backoff Hybrid help to explain the performance differences and is covered in the next section.

3.3.3 Execution Time Breakdown

This section provides analysis of where the PTS and PTS-Backoff Hybrid algorithms spend execution time and a detailed accounting of the overheads incurred. Figure 3.8 shows the overall runtime normalized to a single processor execution and how that time is spent. Figure 3.9 shows the proportion of time spent in each type of execution time category. These categories are the following: 1) Non-Trans - Time spent in user mode that is not inside a transaction or related to scheduling a transaction, 2) Kernel - Time spent operating in the Linux kernel, 3) Trans - Time spent executing in a transaction, this counts both committed and aborted transactions, 4) Abort - Time spent rolling back a transaction's write set, 5) Escape - Time spent suspending a transaction to service PAL code operations, like filling a TLB miss, 6) Sched Begin - Time spent executing the `scheduleTx()` function

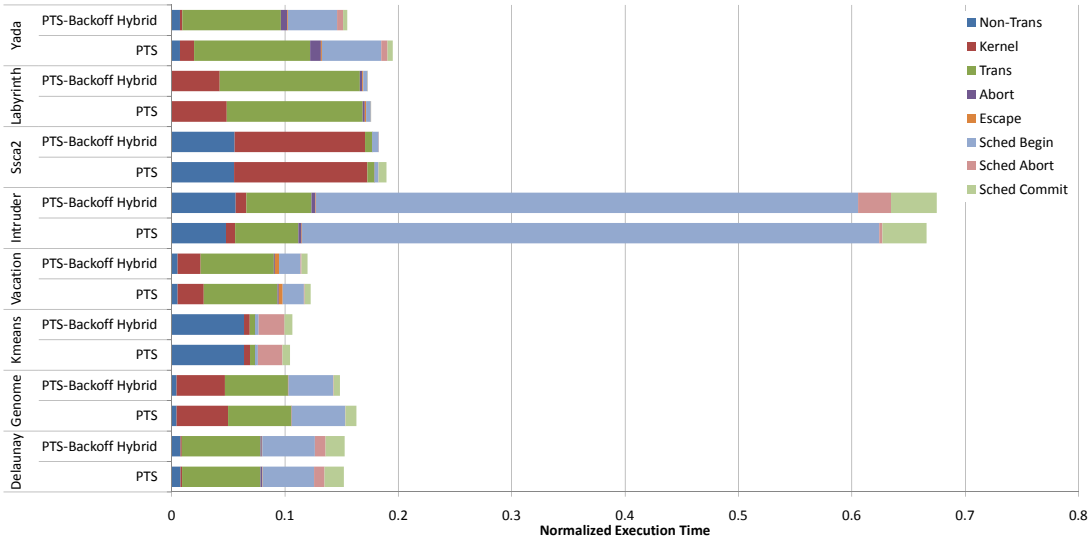


Figure 3.8: Breakdown of where time is spent in the PTS and PTS-Backoff Hybrid predictors normalized to single core performance.

of PTS, 7) Sched Abort - Time spent executing in the `txConflict()` function, 8) Sched Commit - Time spent executing in the `commitTx()` function.

As can be seen in both Figures 3.8 and 3.9 all the benchmarks except for the *Kmeans* and *Ssca2* benchmarks spend a large portion of time executing in transactions as was the original design goal of the STAMP benchmark suite. Both PTS and PTS-Backoff Hybrid techniques spend a fair amount of time in "Sched Begin", indicating that the runtime is spending execution time predicting if execution can proceed. This implies that there is room here for optimization of PTS operations to allow for more performance to be extracted. Hardware acceleration may be valuable for this operation. The *Intruder* benchmark in particular spends almost all of its execution time predicting if a transaction should be scheduled to run. The "Sched Abort" and "Sched Commit" categories are smaller by comparison, indicating that these are fairly well optimized. This trend holds except for in the *Kmeans* benchmark. The reason this benchmark spends more time in "Sched Abort" and "Sched Commit" is because the scheduling functions short circuit the scheduling because it has a large number of small transactions that do not need full scheduling support.

For all the benchmarks except *Yada*, *Genome* and the *Ssca2* benchmarks, PTS and PTS-Backoff Hybrid are almost equivalent in performance. The *Ssca2* benchmark makes sense because there is almost no contention, and therefore PTS is turned off for the entire benchmark, but the cache problems remain. This hurts the performance of *Ssca2*. There is also some load imbalance caused by heavy use of barriers and the fact the system is overcommitted. A non-overcommitted system solves the scaling problems for this partic-

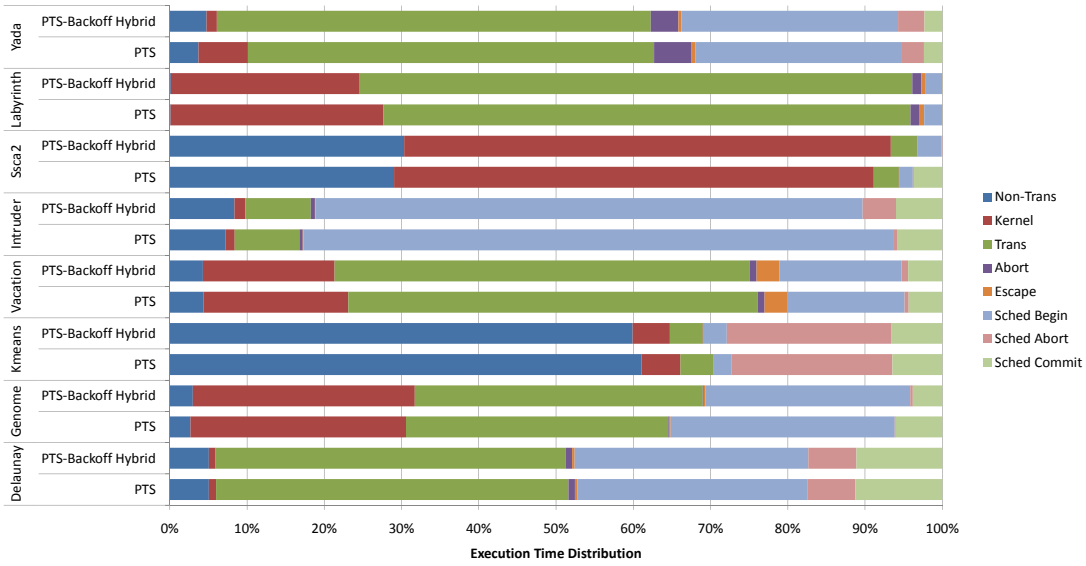


Figure 3.9: Distribution of where time is spent in the PTS and PTS-Backoff Hybrid predictors, each benchmark is normalized to its own runtime.

ular benchmark which can be seen in Appendix A. The *Genome* benchmark gets a large boost when using PTS-Backoff Hybrid over PTS due to spending less time in both "Sched Begin" and "Sched Commit". This is due to at the beginning of the *Genome* benchmark it spends time inserting elements into a hash table. These operations are very parallel and small transactions that do not need the overhead of the full PTS algorithm. During this phase, PTS-Backoff Hybrid keeps PTS turned off until later, allowing better performance from executing less code. The *Yada* benchmark also performs better using PTS-Backoff Hybrid over using PTS. This appears to be happening due to spending less time in the kernel. Because PTS uses `pthread_yield()` to enforce scheduling decisions, this requires a switch into kernel mode to invoke the thread scheduling routines to swap threads. If `pthread_yield()` is used many times, it will affect the amount of time spent in the kernel, and it appears that PTS-Backoff Hybrid is calling `pthread_yield()` less, and is the main cause for better performance along with a lower contention rate. Overall both techniques perform almost the same, and spend time in roughly the same time categories. For a PTS-Backoff Hybrid to be more effective, methods to turn it on and off dynamically during runtime will be required.

Tables 3.5 and 3.6 shows the average number of cycles a transaction spends performing scheduling operations and executing the transaction for PTS and PTS-Backoff Hybrid respectively. The tables also provide the percent overhead PTS incurs per committed transaction. The components that constitute scheduling overhead are the cycles spent in the kernel

	Delaunay	Genome	Kmeans	Vacation
Transactional	1237	361	12	701
Kernel	24	297	15	248
Sched Begin	808	308	7	201
Sched Abort	166	1	62	7
Sched Commit	305	64	19	57
Total	1303	670	103	513
(w/o Kernel)	(1279)	(373)	(94)	(265)
Percent Overhead	105%	186%	858%	73%
(w/o Kernel)	(103%)	(103%)	(733%)	(38%)
	Intruder	Ssca2	Labyrinth	Yada
Transactional	79	12	775063	1466
Kernel	11	238	312659	178
Sched Begin	721	6	26525	747
Sched Abort	3	0	26	78
Sched Commit	55	14	71	66
Total	790	258	339281	1069
(w/o Kernel)	(779)	(20)	(26622)	(891)
Percent Overhead	1000%	2150%	44%	73%
(w/o Kernel)	(986%)	(167%)	(3%)	(61%)

Table 3.5: Amount of scheduling overhead experienced in cycles per transaction commit for PTS.

due to calls to `pthread_yield()`, and cycles spent executing the PTS runtime functions. Numbers are presented that include both cycles in the kernel and cycles of overhead with the kernel cycles omitted because in the case of *Ssca2* the kernel time is caused primarily by load imbalance. For both PTS and PTS-Backoff Hybrid the amount of overhead incurred is fairly high. In many cases the amount of time spent performing contention management operations is greater than the average time spent executing a transaction. The highest seen is in the *Intruder* benchmark where the benchmark spends 10x more time performing contention management, but as seen this allows for the program to maintain better performance than Backoff. Other benchmarks see the cost of scheduling to be between 50% and 200% of the execution time of a transaction. This is still very high in terms of overhead cost, and reducing the amount of time spent performing PTS operations is desirable.

3.3.4 Sensitivity Studies

To better understand the effect of the different optimizations and parameters available to the PTS and PTS-Backoff Hybrid contention managers a series of sensitivity studies are performed to better understand the effects. These studies include looking at the sensitivity

	Delaunay	Genome	Kmeans	Vacation
Transactional	1236	361	13	696
Kernel	23	278	14	220
Sched Begin	825	257	9	205
Sched Abort	170	3	65	11
Sched Commit	302	37	20	57
Total	1320	575	108	493
(w/o Kernel)	(1279)	(297)	(94)	(273)
Percent Overhead	107%	159%	831%	71%
(w/o Kernel)	(105%)	(82%)	(723%)	(39%)
	Intruder	Ssca2	Labyrinth	Yada
Transactional	81	12	798475	1253
Kernel	13	233	273098	29
Sched Begin	677	11	23499	625
Sched Abort	41	0	491	76
Sched Commit	57	0	16	51
Total	788	244	297104	781
(w/o Kernel)	(775)	(11)	(24006)	(752)
Percent Overhead	973%	2033%	37%	62%
(w/o Kernel)	(957%)	(92%)	(3%)	(60%)

Table 3.6: Amount of scheduling overhead experienced in cycles per transaction commit for PTS-Backoff Hybrid.

of the PTS managers to the "Small Transaction" short circuiting optimization, the "Split Transaction ID" optimization for applicable benchmarks, Bloom filter size sensitivity and finally how the benchmarks behave if perfect hardware could be constructed to determine latency sensitivity of PTS.

3.3.4.1 Small Transaction Prediction Optimization

The small transaction prediction optimization was implemented to allow small transactions to short circuit the scheduling algorithm to reduce overhead and allow more optimistic predictions for small transactions. Figure 3.10 shows the performance differences in terms of overall speedup for a 16 core system using PTS that has no such optimizations applied (neither the small tx optimization nor the split ID optimization). The *Kmeans* benchmark sees the biggest gain from using the small transaction optimization, improving from a speedup of $\sim$7x to close to a speedup of $\sim$10x on a 16 processor CMP. The other benchmarks see limited gain from this optimization. The *Ssca2* benchmark does show some gain, but the difference is not nearly as large as than seen by the *Kmeans* benchmark. Overall, PTS is rather insensitive to this optimization unless the benchmark runtime is dominated

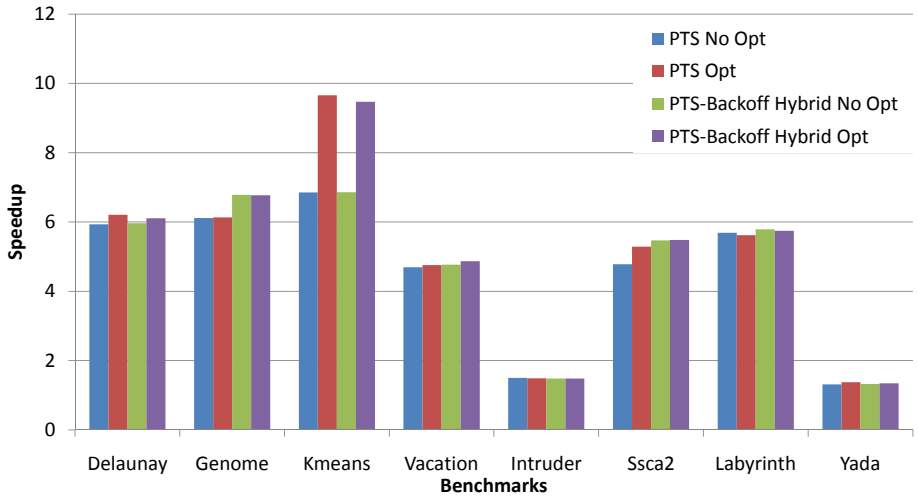


Figure 3.10: Speedup of PTS and PTS-Backoff Hybrid using the Small Transaction Optimization compared to No Optimization for a 16 processor system.

by small transactions that have some contention that would benefit from cutting down on overhead as much as possible such as *Kmeans*.

3.3.4.2 Splitting Transaction ID

The second optimization that was covered in Section 3.2.3 was allowing a single transaction to use more IDs to approximate the act of resetting the confidence of conflict between other transactions in an effort to optimize in the case of transient conflicts. This optimization was applied only to the *Delaunay*, *Vacation* and *Yada* benchmarks as they were the only applications to show this tendency from experimentation and inspection of the code. Figure 3.11 shows the sensitivity of these three benchmarks to this optimization when compared to just PTS using the small transaction optimization. As can be seen this gets gains on all three benchmarks, the most significant gains are seen for the *Vacation* and *Yada* benchmarks. For the *Vacation* benchmark, this optimization almost doubles its performance. This happens because using extra transaction IDs allows PTS to capture the conflict patterns better as the Red-Black Tree evolves during the execution of the benchmark. *Yada* sees an even greater improvement, improving from practically serial execution for both PTS and PTS-Backoff Hybrid to speedups of $\sim$5x and $\sim$6x respectively. This shows that the PTS technique is very sensitive to transactions that have transient conflicting behavior. It can be overly pessimistic in these cases. Still, this method of splitting transaction IDs is non-portable. In the next Chapter I present new methods that can deduce this behavior automatically.

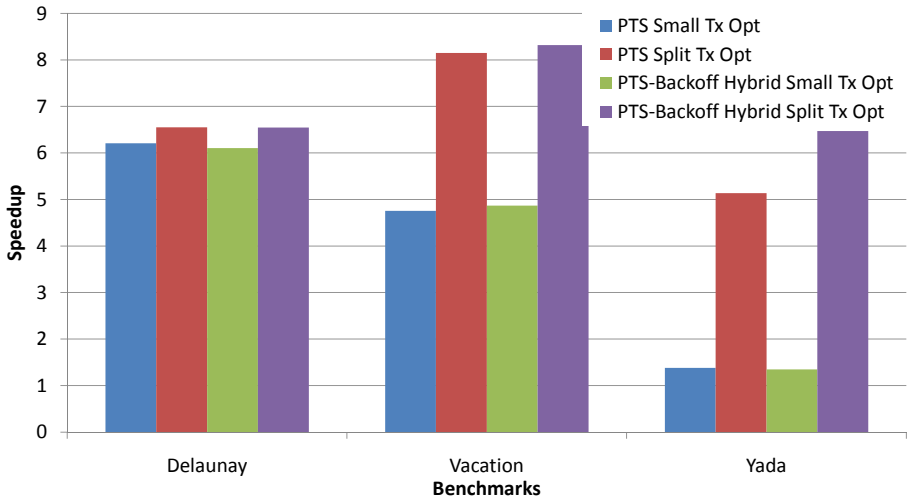


Figure 3.11: Speedup of PTS and PTS-Backoff Hybrid using the Split Transaction ID optimization over Small Transaction Optimization for the Delaunay, Vacation and Yada benchmarks.

3.3.4.3 Bloom Filter Size Sensitivity

Another test performed was to measure the sensitivity of prediction to the Bloom filter sizes used. I measured sensitivity from a size of 512bit to 8192bit for the Bloom filters. The results are presented in Figures 3.12 and 3.13 for the PTS and PTS-Backoff Hybrid predictors respectively.

For PTS, it shows a very clear sensitivity to Bloom filter size. But, each benchmark is affected differently depending on its characteristics. The *Delaunay* benchmark prefers larger bloom filters, and performance steadily increases as larger filters are used. The *Delaunay* benchmark benefits from improved predictions and can amortize the cost of manipulating a large 8192bit Bloom filter. For the *Genome* benchmark, it sees a gradual increase in performance until a Bloom filter size of 2048bits and then performance declines as the larger Bloom filters cost more in terms of execution time to manipulate. For the *Kmeans*, *Vacation*, *Intruder* and *Ssca2* the smallest Bloom filter size of 512bits performs the best. This is because these benchmarks are very sensitive to scheduling overheads incurred by PTS. The *Labyrinth* and *Yada* benchmarks appear to be mostly insensitive to Bloom filter size and no one size appears to be the absolute best. Both of these benchmarks have large transaction sizes that can amortize the cost of doing the Bloom filter manipulations.

For PTS-Backoff Hybrid the picture is a little different when the contention management schemes are able to switch between Backoff and PTS. The *Genome* benchmark shows a less than clear trend. The same happens with the *Vacation* benchmark. This is due to the

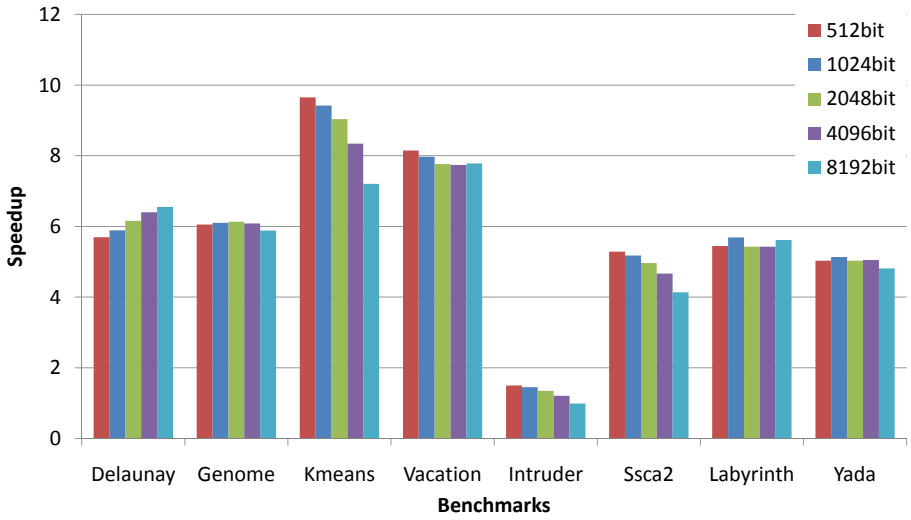


Figure 3.12: Sensitivity to Bloom filter size for PTS for a 16 processor system for best performing configuration.

effects of delaying switching to PTS, and the prediction confidence values are changing slightly to affect performance. Other trends remain unchanged, the *Delaunay* benchmark still prefers the largest Bloom filter that is available, the *Kmeans* and *Intruder* benchmarks still want to use the smallest Bloom filter to reduce overhead as much as possible.

One of the concerns with this sensitivity test is the benchmarks disagree over the what the optimal Bloom filter size is. Some applications prefer a large Bloom filter while some want the smallest Bloom filter available. The performance difference can be quite large, like in the *Kmeans* benchmark, performance goes from ∼9.5x speedup with a 512bit Bloom filter to a ∼7x speedup if a 8192bit Bloom filter is used. This implies a Bloom filter needs to ideally be variable in size. This should be possible, but a solution to this is not addressed in this thesis and is assumed to be available throughout the rest of this document.

3.3.4.4 Sensitivity to Prediction Overhead

One of the shortcomings of PTS and PTS-Backoff Hybrid is that the technique is implemented entirely in software. As seen in Section 3.3.3, the overhead imposed by PTS can be very high on a per transaction basis, in some cases adding 2x-10x more cycles to the execution time of the transaction. A final sensitivity experiment I ran was to test the sensitivity of the benchmarks to prediction latency, e.g. the time it takes to make a decision whether to suspend or execute and also the time it takes to manipulate the bloom filters on commit. The execution time of doing a `pthread_yield()` still remains as well as time to stall a transaction as there is no way to eliminate accounting for serialization operations.

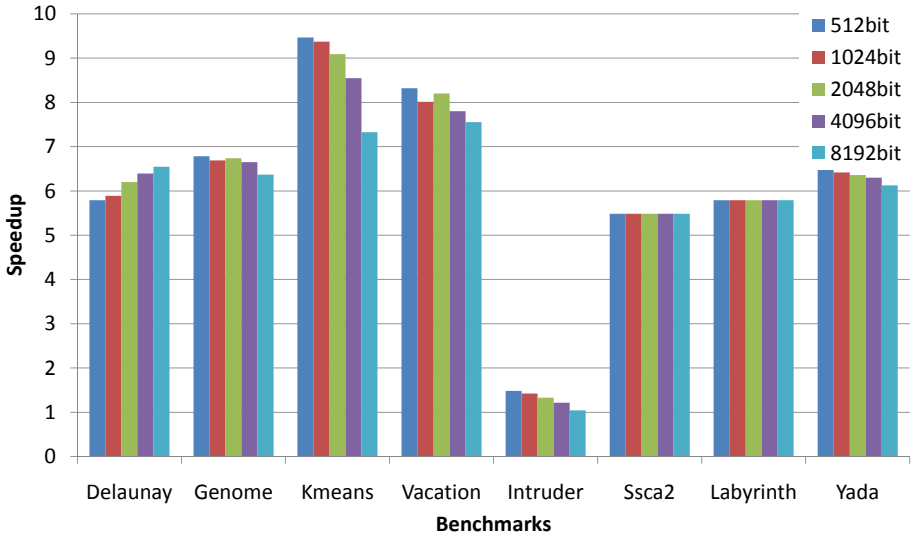


Figure 3.13: Sensitivity to Bloom filter size for PTS-Backoff Hybrid for a 16 processor system for best performing optimizations.

To perform these experiments, M5 was modified to perform all the scheduling operations in "magic" hardware that took a pre-specified amount of cycles to complete. The experiments were then performed for scheduling operations to take between 1 and 10,000 cycles. The results for each STAMP benchmark are presented in Figure 3.14. For the *Kmeans*, *Ssca2* and *Intruder* benchmarks they are very sensitive to prediction latency and prefer the fastest prediction latencies as shown by the monotonic decrease in performance and latency increases. This makes sense as these three benchmarks had scheduling add large amounts of overhead to the transactional execution as documented in Tables 3.5 and 3.6. The *Vacation* and *Yada* benchmarks show similar behavior, they want as fast a scheduling prediction as possible, but the performance penalty is less on average. This shows for these three benchmarks they are more tolerant to overhead incurred by scheduling. The last two benchmarks *Delaunay* and *Genome* show interesting behavior. These benchmarks exhibit non-monotonic behavior as the latency of performing PTS operations increases. The performance at 1000 cycles latency is better than the performance at 1 cycle. At 10,000 cycles, the prediction overhead dominates and performance falls off as it does with the other benchmarks. The reason for this non-monotonic behavior is observed is from overhead induced by repeated calls to `pthread_yield()`, and the predictions being slightly more optimistic due to the extra latency incurred. These two benchmarks have both high contention and a fair amount of large transactions that force calls to `pthread_yield()`. In the case where the prediction latency is fast, the calls to `pthread_yield()` happen rapidly. This causes an increase in kernel time and therefore a drop in performance. When predic-

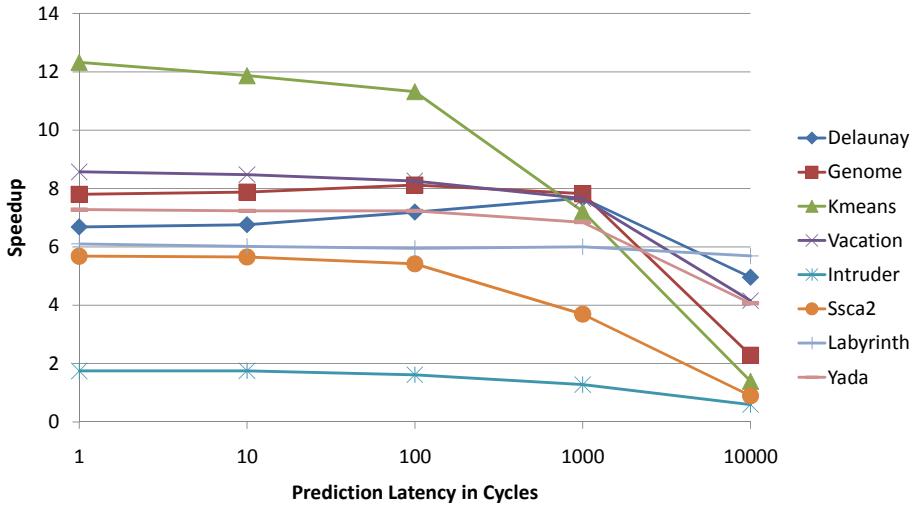


Figure 3.14: Sensitivity to PTS scheduling latency for a 16 processor system.

tion latency is higher, there are less calls overall to `pthread_yield()` reducing overhead. The predictions are also more optimistic at higher latencies because the prediction routines are using slightly stale view of the system to make predictions. Both these factors work together to cause the non-monotonic behavior seen in these benchmarks.

3.3.5 Measuring Prediction Accuracy

Prediction accuracy would be a very useful metric to derive and use in analyzing the performance of the PTS algorithm. The prediction accuracy would involve first determining the perfect ordering for transactions to execute in to serve as an oracle. This would require running all possible interleavings of transactions to determine such an oracle schedule. For benchmarks like *Delaunay* that have many patterns of contention, this would be impossible to determine as the search space would be very large. This problem is similar to the Traveling Salesman problem where transactions would be cities and one has to find the fastest way to execute all transactions. There is no way to determine if a locally optimal schedule of transactions is globally optimal, just as in the Traveling Salesman problem. The Traveling Salesman problem is provably NP-Complete, and therefore so is the problem of finding the best possible execution schedule of transactions. Ultimately it is not possible to truly know how good the predictions are. It can be inferred from the performance and measured contention how well predictions are being made relative to other contention management techniques though.

It is possible to characterize qualitatively the prediction accuracy of PTS, as was done in [27]. Table 3.7 shows the percentages of misspredictions as viewed from the point-

Percent Misspredict	Delaunay	Genome	Kmeans	Vacation
Total	47.6%	37.8%	7.6%	29.4%
Parallel	25.0%	2.3%	3.5%	9.9%
Serial	22.6%	35.5%	4.2%	19.5%
Percent Misspredict	Intruder	Ssca2	Labyrinth	Yada
Total	32.4%	0.1%	12.3%	40.2%
Parallel	4.6%	0.1%	11.3%	29.2%
Serial	27.9%	0.0%	1.0%	11.0%

Table 3.7: The PTS technique's prediction accuracy, as measured from the point-of-view of the algorithm.

of-view of the predictor. What this means is Table 3.7 shows the breakdown of how the predictor is evaluating its edge weightings from the dynamic conflict graph. This was measured by letting the PTS algorithm run entirely as a zero latency module in M5, and recording all predictions made for a transaction, and then evaluating those predictions when the transaction conflicts or commits. For example, if Tx1 predicts it must serialize against Tx2 and Tx3, on commit it will intersect its RWSet with those transaction's saved RWSets and determine if the serialization was predicted correctly from the point-of-view of the algorithm. This would correspond to the *Percent Misspredict Serial* row in the Table. If a transaction has a conflict, then it is a misspredicted parallel transaction and is classified in the *Percent Misspredict Parallel* column. As can be seen in the table, transactions have a large mix of predicting too parallel or too serial. For example, the *Delaunay* benchmark misspredicts on the parallel side, whereas the *Vacation* benchmark misspredicts on the serial side. In the next chapter, I will present techniques that aim to fix these problems as well as eliminate all the application specific optimizations presented in this chapter.

3.4 Conclusions

This chapter presented the "Proactive Transaction Scheduling" (PTS) method for managing contention in an Eager/Eager LogTM like HTM. It establishes that reactive contention managers such as Randomized Linear Backoff do a poor job at managing high contention situations through empirical studies. It then shows that transactions in the STAMP benchmark suite exhibit "conflict locality", and therefore future conflicts can be being predicted from the observation that conflicts seen in the past are likely to happen in the future. This chapter then presented the implementation of a software runtime that implemented

PTS along with various optimizations that can be applied to solve bottlenecks such as over-pessimistic predictions due to transient conflict behavior and high overhead of PTS as seen by small transactions. The results presented show that PTS can attain on average a 2x improvement over a backoff based contention manager.

CHAPTER 4

Bloom Filter Guided Transaction Scheduling

This chapter presents the published work "Bloom Filter Guided Transaction Scheduling" (BFGTS) [28]. BFGTS is a generalized transaction scheduling contention manager that better predicts future conflicts over the previously presented PTS technique from the previous chapter. This chapter also presents hardware acceleration and ISA extensions that can speed up prediction operations to further extract performance. This chapter differs from the previously published work by offering a more detailed implementation section. The evaluation is also more detailed over the published work, as well as presenting more sensitivity studies to give further insight into the workings of the BFGTS algorithm.

4.1 Motivation

In Chapter 3 I presented the "Proactive Transaction Scheduling" technique. In that chapter I showed that it is a substantially better alternative (getting on average 2x performance improvement) for reactive contention managers for an Eager/Eager HTM that experiences high contention.

There were still multiple drawbacks to the PTS contention manager. One was its high overhead due to being an all software technique. After analyzing PTS, it will be shown that accelerating the operations for scheduling a transaction would be beneficial. This acceleration would allow me to eliminate the need for short-circuiting predictions made for small transactions. The other main drawback to PTS was its over pessimism in transient conflict cases. PTS had no way to identify how transactions are acting that could affect how optimistic or pessimistic one could be when predicting future conflicts. This led to the development of the "Split Transaction" optimization to allow for more optimistic predictions in the face of transient conflict behavior. Unfortunately this optimization is not portable to other applications as it is application specific and required deep understanding of the

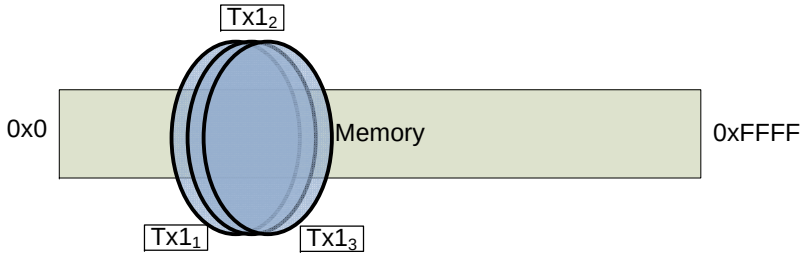


Figure 4.1: Example transaction executions that show similar execution behaviors over time.

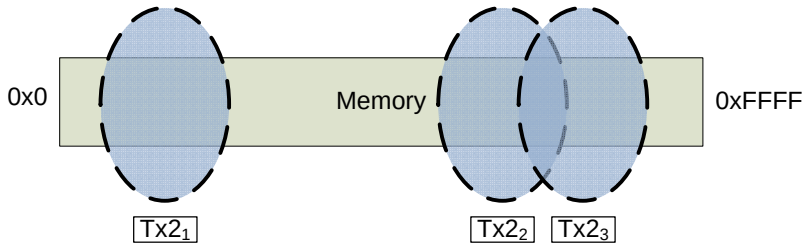


Figure 4.2: Example transaction executions that show dissimilar execution behaviors over time.

application's behavior. PTS was the proof of concept to prove that transaction scheduling could be accomplished efficiently and be effective over simpler techniques. This chapter will present more advanced techniques that capture the above behavior automatically and lead to an overall better performing generalized transaction scheduler.

4.1.1 Transaction Behavior and defining the Similarity Metric

The dynamic nature of code executed inside a transaction makes it hard to predict good schedules that avoid conflicts. Just tracking conflict history or contention rate is not enough to get a full picture of the program's behavior. To form good scheduling decisions a proactive scheduler needs to be able to identify the behavior of transactions and how they are being affected by conflicts. This can help guide a scheduler to be more optimistic scheduling some transactions while scheduling pessimistically for others.

Take the following synthetic example. Assume a group of transactions are modifying locations in memory. Some transactions continually modify the same general locations in memory each time they are executed, shown in Figure 4.1. This transaction exhibits a high amount of memory locality on each consecutive execution. For the rest of this paper we will term this locality property "*Similarity*". Other transactions may jump around, working in

different regions of memory each time they execute, shown in Figure 4.2. This transaction exhibits low similarity on consecutive executions. In terms of transaction conflicts, if two transactions having low similarity conflict in the past, this conflict is likely to be transient, e.g., inserting to a hash table. Conversely, if two transactions conflict and they have high similarity, this conflict is likely to persist, e.g., enqueuing and dequeuing from a queue. This property can help a scheduler identify such behaviors and treat conflicts accordingly. As seen in the sensitivity studies presented in Chapter 3, application specific optimizations were made by splitting transaction IDs that take advantage of low similarity behavior to improve performance of PTS. Similarity is defined in this thesis as a value between 0 and 1 as follows:

$$Similarity = \frac{SetSize(RWSet_{t-1} \cap RWSet_t)}{MAX(SetSize(RWSet_{t-1}, SetSize(RWSet_t)))} \tag{4.1}$$

SetSize counts the number of entries in a set, $MAX(SetSize(RWSet_{t-1}), SetSize(RWSet_t))$ takes the largest number of entries in either transaction's read/write set, and *RWSet* is the set of addresses touched by the transaction. Similarity in this case is calculated using the just completed transaction from time t and the previous execution $t-1$ and using the maximum of the set sizes will always restrict the value to be between 0 and 1. The more entries in common between two $RWSet's$ and if the two transactions have similar RWSet sizes, the higher the similarity (closer to 1). Similarity not only captures the similarity in memory regions touched by two transactions but also captures how the RWSet size changes. Looking at Figure 4.1, the similarity for $Tx1$ would be close to 1 as each consecutive execution touches similar memory and the ovals (representing RWSet size) are of similar size.

This type of behavior was measured in the STAMP benchmarks. Table 4.1 shows the conflict graph for each transaction, and the measured value of each transaction's *similarity*. The *Conflict Graph* in Table 4.1 is a matrix representation of the conflict graph seen by transactions during the execution of the STAMP benchmarks. Each number in the column *Conflict Graph* represents a conflict that occurred at some point between the row transaction ID and the transaction ID listed in column *Tx*. Each transaction ID represents a transaction defined in the code. For the Delaunay benchmark, it has transactions that conflict with every other transaction in the system. The transactions `0`, `2` and `3` have a high similarity and should be serialized, while the transaction `1` has a very low similarity and should be treated by a scheduler as a transaction that has transient conflicts. A scheduler that can better identify transaction behavior will be able to make more informed scheduling decisions. As seen in Table 4.1 there is wide range of conflict behavior for the STAMP benchmark suite, both in transactions conflict sets and the similarity observed by each transaction.

Benchmark	Tx	Conflict Graph	Sim	Benchmark	Tx	Conflict Graph	Sim
Delaunay [72]	0:	0 1 2	0.64	Intruder	0:	0	0.67
	1:	0 1 2 3	0.04		1:	1 2	0.40
	2:	0 1 2 3	0.56		2:	1 2	0.66
	3:	1 2 3	0.90				
Genome	0:	0	0.12	Ssca2	0:	0	0.90
	1:		0.25		1:		0.90
	2:	2 3	0.65		2:	2	0.57
	3:	2	0.74				
	4:	4	0.29				
Kmeans	0:	0	0.38	Labyrinth	0:	0	0.86
	1:	1 2	0.67		1:	1 2	0.45
	2:	1	0.68		2:	1 2	0.90
Vacation	0:	0	0.26	Yada	0:	0 2 4	0.57
					1:	2	0.5
					2:	0 1 2 3 4 5	0.30
					3:	2	0.5
					4:	0 2 4 5	0.52
					5:	2 4 5	0.90

Table 4.1: Matrix representation of the conflict graph observed during the execution of each STAMP benchmark and measured average similarity for each unique transaction.

4.1.2 Bloom Filter Operations to Extract Similarity

As seen in the previous section, calculating similarity requires a set intersection, which can be expensive if the sets are compared pairwise. This is not feasible in an HTM, so Bloom Filters [32] are used to represent and manipulate transaction read/write sets efficiently as was done in Chapter 3. As shown by Sanchez et al. [95] implementing bloom filters can be done efficiently in hardware. A unique contribution of this work is to develop Bloom filter manipulations to estimate similarity efficiently and apply this property to a generalized transaction scheduler. The equations below delineate how similarity is calculated and maintained in this work as there are many possible ways to do so.

Work by Michael et al. [83] is used to develop the Bloom filter operations to estimate similarity. The Bloom filter manipulations were originally developed for fast join operations in large distributed databases. The main equations used for this work are the set size estimations(denoted as $S^{-1}(t)$) of encoded Bloom filters(denoted as $S(t)$). Equation 4.2 calculates the set size estimation of an encoded Bloom filter where t is the number of bits

set, m is the total size in bits of the Bloom filter, and k is the number of hash functions used.

$$S^{-1}(t) \cong \frac{\ln(1-\frac{t}{m})}{k*\ln(1-\frac{1}{m})} \tag{4.2}$$

An estimation of set size of the intersection between two Bloom filters shown in Equation 4.3 is used to estimate how many entries are in common between the two Bloom filters.

$$S^{-1}_{AND}(t) \cong S^{-1}_{1}(t) + S^{-1}_{2}(t) - S^{-1}(S_1(t) \cup S_2(t)) \tag{4.3}$$

Instead of using the $MAX(SetSize(RWSet_{t-1}, SetSize(RWSet_t))$ denominator to calculate similarity as done in Equation 4.1, a the weighted average of RWSet sizes over time was used. The weighted average is quick to calculate and is a good enough approximation for the purposes of the described implementation if the weights are picked properly (in this work, the weight is 0.5). Since the weighted average will be smaller than the $MAX(SetSize(RWSet_{t-1}, SetSize(RWSet_t))$ it is biased towards higher similarity if the RWSet sizes are fluctuating, otherwise the average will not be biased either way significantly. Equation 4.4 defines how BFGTS defines average read/write set size.

$$AvgRWSetSize(Tx_n) = weight * AvgRWSetSize(Tx_{n-1}) + (1-weight) * RWSetSize(Tx_n) \tag{4.4}$$

Equation 4.3 derives the *"Similarity"* metric using Bloom filters to represent read/write sets.

$$Similarity_{t,t-1} \cong \frac{S^{-1}_{AND}(t)}{AvgRWSetSize(Tx_n)} \tag{4.5}$$

Equation 4.6 shows how average similarity is calculated as a weighted average to be stored for use with the BFGTS algorithm.

$$AvgSimilarity_t = (1-weight) * AvgSimilarity_{t-1} + weight * Similarity + t, t-1 \tag{4.6}$$

4.2 Implementation

This section describes the hardware and software implementation of BFGTS. The design uses fine-grained scheduling between transactions and borrows concepts from PTS as presented in Chapter 3. BFGTS maintains a graph structure in software of nodes and edges to represent conflict history and confidence in future conflicts to facilitate scheduling decisions. The majority of BFGTS is implemented as a software runtime that sits between the application and the Operating System. A small TLB like hardware accelerator is also

present that operates when it sees a `TM_BEGIN` instruction from the processor.

During the discussion of BFGTS there are two types of transaction IDs (TxID) that will be used throughout this chapter: *"Static Transaction ID"*(sTxID) and *"Dynamic Transaction ID"*(dTxID). An sTxID is statically assigned in the program code. A dTxID is a concatenation of thread ID and sTxID.

4.2.1 Scheduling Hardware Accelerator

In BFGTS, before a transaction begins execution it must scan a global array called the *CPU Table*—a list of the transactions running on the processors in the system. At each entry in the table, a confidence value representing the likelihood of a conflict between the transaction to be scheduled and the running transaction is retrieved from a global graph data structure which is represented as a 2D matrix in memory. If the confidence exceeds a threshold value the transaction is serialized. Scanning the *CPU Table* at the start of every transaction adds a large amount of overhead to each transaction as shown in the evaluation section in Chapter 3.

BFGTS minimizes this overhead by using a hardware accelerator to scan the *CPU Table*, look up confidence values, and compare them to a preset threshold. This is then used to effect a scheduling decision in a few cycles. This accelerator is triggered upon seeing a `TX_BEGIN` instruction. These operations are relatively simple, therefore the hardware is small. The hardware implements the algorithm shown in Example 4.1.

The scheduling hardware is illustrated in Figure 4.3. It consists of a small cache, a handful of control registers, and logic connected to the coherent interconnect. Each processor gets an identical predictor unit so the predictions are fully distributed. The control registers consist of a CPU Table that represents all the remote processors in the system and the dTxID of its currently executing transaction. The other registers are as follows: a physical base address of the confidence value table to index into, the confidence threshold to compare confidence values against, a shift register for truncating dTxIDs in the CPU Table to sTxIDs, and a register to hold the dTxID of a transaction to serialize against for later access by software.

The hardware predictor contains a small cache that is exclusively used for caching the confidence table. The cache is necessary because the confidence tables can be pushed out of the L1 caches, increasing the time it takes to make a prediction. The cache is also modified to fetch cache lines evicted by an invalidate snoop. This is required to prevent always taking a miss when accessing the cache because the main processor writes to the confidence tables frequently, invalidating entries in the accelerator's cache. The hardware overhead of the small cache and accelerator is very small. This additional hardware is

Example 4.1 Lookup algorithm implemented by hardware accelerator for BFGTS algorithm

```
1  bool scheduleTx(int sTxID)
2  {
3   for (i = 0; i < sizeof(CPUTable); i++){
4    confidx = CPUTable[i] >> shift_value;
5    conf=confidenceTable[sTxID][confidx];
6    if (conf > threshold) {
7     dTxID_wait_on = CPUTable[i];
8     return true; //conflict predicted
9    }
10  }
11  return false; //no conflict predicted
12 }
```

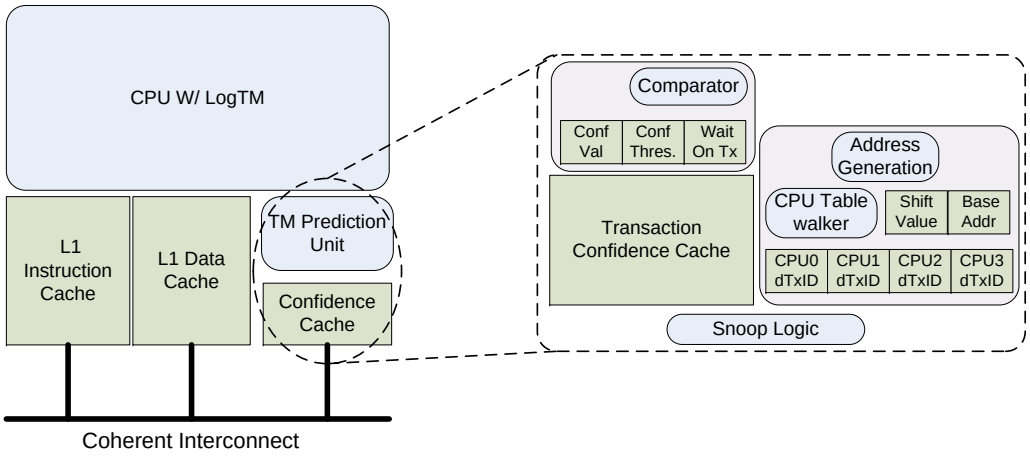


Figure 4.3: Hardware required to accelerate scheduling on TX_BEGIN for a 4 core system.

very similar to having an additional TLB added to the processor as it implements a small hardware table walking engine and a cache to store entries.

To interface the software with the hardware prediction unit, the `TX_BEGIN` instruction is modified to trigger the predictor to form a prediction. The `TX_QUERY_PREDICTOR` instruction is added to modify the control registers of the predictor. `TX_BEGIN` traditionally puts the CPU into transactional mode, takes a register checkpoint, but takes no register arguments. `TX_BEGIN` now takes a vector to a `suspendTx()` function for the processor to jump to if the hardware predictor returns that a conflict is likely and the transaction should serialize. `TX_BEGIN` triggers the hardware predictor to perform the algorithm in Example 4.1 and waits for it to return either yes, a conflict is likely and jump to `suspendTx()` or continue execution of the transaction. The `TX_QUERY_PREDICTOR` instruction acts like the `ioctl()`

system call for the accelerator engine. The instruction is used to communicate information such as the physical address of the confidence table for the hardware predictor to use for looking up confidences, query what dTxID to serialize against, set the confidence threshold to use, and query if a dTxID is still executing in the system to allow busy waiting on small, short running transactions.

BFGTS, like PTS presented in the previous chapter requires additional requests to be added to the coherent interconnect to allow the predictors to update their arrays representing the state of each remote CPU. When a transaction is allowed to execute, it broadcasts onto the interconnect the dTxID of the starting transaction as well as the CPU ID. The other predictors snoop this broadcast and update their arrays accordingly. This is similar to TLB shootdown mechanisms when page table structures are updated on one CPU that need to be updated to other CPU's TLBs. On a transaction commit or abort, the CPU broadcasts the CPU ID along with the transaction outcome for the other predictors to update their internal state. The PTS out-of-band information is still present, specifically the returning of the dTxID of the remote conflicting transaction that caused a local transaction to abort.

4.2.2 Software Implementation

The remainder of BFGTS is kept in a software runtime to do book keeping operations, such as updating the confidences and calculating similarities of transactions. These book keeping functions can be quite complicated. Therefore a hardware mechanism would be infeasible as the amount of logic and storage required would be on the order of an additional processor core. Properly optimized software with the necessary ISA support is sufficient to implement this part of BFGTS and still get acceptable performance.

4.2.2.1 Data Structures

The data structures used in BFGTS are inspired by PTS, but modified to be more efficient in both layout and space. An overview of the data structures are shown in Figure 4.4.

The first data structure is a set of confidence tables that are allocated per processor. This allows easy caching in the private cache attached to the hardware accelerator. The confidence tables hold the values that predict how likely a conflict is between two transactions if they were to execute concurrently in the future. In PTS, the confidence table was one global table that had a confidence entry for each dTxID. This table could grow to be 10's of MBs in size. Instead of tracking a confidence for every pair of dTxIDs, BFGTS compresses the table to only maintain confidence values between each pair of sTxIDs assigned in the code as well as making private tables per processor. By tracking only sTxIDs, the

confidence table reduces to a maximum size of *800Bytes* for the benchmarks tested. This is substantially smaller as well as making it more easily cacheable per processor.

The second data structure required is an array of statistics kept for each dTxID that is encountered during runtime. For each dTxID three items are stored: average transaction size, similarity, and if a conflict was predicted, the dTxID of the predicted conflicting transaction. The final data structure is a table of the most recent Bloom filters for each dTxID. The Bloom filters are used to calculate the average similarity of each dTxID, and to update confidence of conflict between sTxIDs on commit.

These data structures grow in similar fashion to the data structures of PTS. The *Confidence Tables* grow in memory in $O(M^2)$ where M is the number of transactions declared statically in the code. The *Tx Statistics Array* grow in memory in $O(NM)$ where N is the number of threads and M is the number of transaction declared in the code. These structures can grow to be unbounded, and therefore may be infeasible for very large transactional codes. A solution to this may be to allow aliasing in the prediction data structures—multiple transactions mapping to the same *Confidence Table* and *Tx Statistics Array* locations. This is left as future work and not explored in this thesis.

4.2.2.2 Scheduling Sub-routines

The bulk of BFGTS exists as a software runtime. The software executes in user space, and is fully distributed. Three scheduling operations are done in software: Transaction Suspend, Transaction Abort, and Transaction Commit.

Before covering the scheduling operations, it is important to cover more in depth how the software coordinates with the hardware accelerator presented in the previous section. To keep the hardware accelerator as simple as possible, the software is used to set up the hardware before each transaction. To do this the software performs the operations shown in Example 4.2. These operations consist of pre-calculating the row in the *Confidence Table* matrix the predictor will be indexing, as the predictor only has a simple address generation unit that adds an offset (an sTxID in this case) to a base address. The base address is then passed to the `XACTION_QUERYPRED` instruction which uses the passed in command along with the base address and transforms it into a physical address for the predictor to use.

Transaction Suspend is the routine that the CPU vectors to when the `TX_BEGIN` instruction is informed by the hardware predictor a conflict is likely and the transaction needs to serialize. Example 4.3 illustrates how predicted conflicts are serialized in BFGTS.

The dTxID of the transaction being serialized against is recorded for use later during transaction commit. If a $dTxID_i$ is predicted to conflict with another $dTxID_j$ that is historically small, then $dTxID_i$ stalls waiting for $dTxID_j$ to commit or abort. If $dTxID_j$ is larger

Example 4.2 Predictor Hardware setup Pseudo Code for BFGTS algorithm

```
1 void predictorSetup(int cpuID, int dTxID)
2 {
3  sTxID = dTxID >> shift_value;
4  confTableBase = &conf_table[cpuID][sTxID*num_txs];
5
6  XACTION_QUERYPRED(confTableBase,
7                    SETCONFTABLE_BASEADDR);
8 }
```

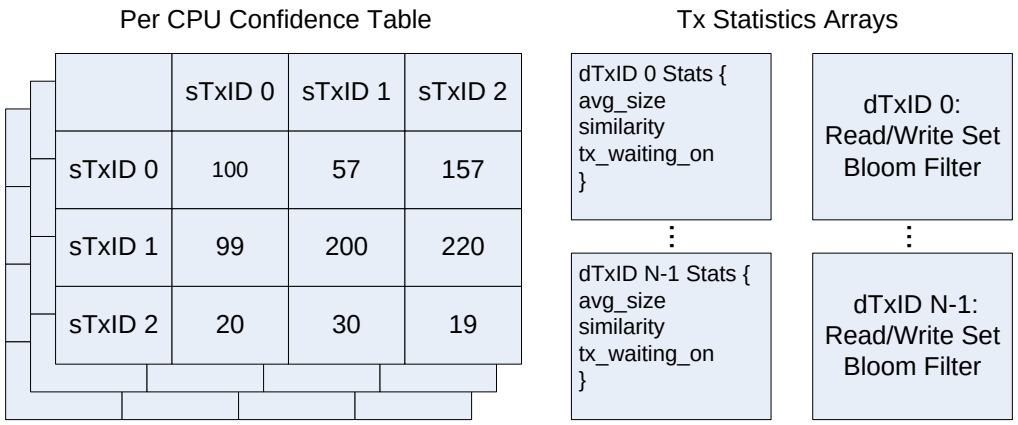


Figure 4.4: Data-structures for the confidence tables, transaction statistics table and Bloom filter tables kept in virtual memory.

Example 4.3 Suspend Transaction Handling Pseudo Code for BFGTS algorithm

```
1 void suspendTx(int dTxID, int dTxIDSusp)
2 {
3  sim=0.5 * (simOf(dTxID) + simOf(dTxIDSusp));
4  decay=decayVal * (1 - sim);
5  decConfProb(sTxID, sTxIDSusp, decay);
6  statsTable[dTxID].txWaitingOn = dTxIDSusp;
7  if(avgTxSize(dTxIDSusp) >= SMALL_TX_SIZE){
8   pthread_yield();
9  } else {
10  stallOnTx(dTxIDSusp);
11 }
12 restore_checkpoint();
13 }
```

than a small transaction threshold then it is suspended and another thread switched in. As in PTS, BFGTS is taking advantage of an overcommitted system to attempt to hide serial-

Example 4.4 Conflict Handling Pseudo Code for BFGTS algorithm

```
1 void txConflict(int dTxId,int dTxIdConf)
2 {
3  sim = 0.5*(simOf(dTxId) + simOf(dTxIdConf));
4  inc = incVal*sim;
5  incConflictProb(dTxID, dTxIDConf, inc);
6  incConflictProb(dTxIDConf, dTxID, inc);
7 }
```

ization latency by switching in a different thread to accomplish other non-conflicting work. In BFGTS `pthread_yield()` is used to switch threads. Upon exiting `suspendTx()` the transaction restores its register checkpoint and jumps to the PC to re-execute the `TX_BEGIN` instruction. A problem noticed with PTS as seen in the previous chapter is that it could be too pessimistic at times, due to the probability of Bloom filter intersection returning true is relatively high with a moderate number of addresses hashed into the filter. This meant that on transaction commit, the Bloom filter intersection used to either increment or decrement confidence is more biased to incrementing confidence. This reduced how much parallelization could be exposed. To allow transactions to return to scheduling optimistically, a *decay* operation is used to slowly decrease the confidence that a conflict will occur between two sTxIDs in BFGTS. Decay is weighted by the average similarity of the two dTxIDs that are predicted to conflict to drive how quickly decay occurs. If a conflict is predicted between two transactions and they are both very similar to themselves, then a predicted conflict is likely to be accurate, and the decay is small. On the other hand, if the transactions are dissimilar, the decay will be large, allowing the confidence to decay quickly to allow the two transactions to be scheduled concurrently.

On *Transaction Abort* due to a conflict, first the transaction rolls back its speculatively written state. Then it calls the `txConflict()` routine presented in Example 4.4 to increment confidence values of future conflict between the two dTxIDs. Again similarity is used to guide how much the confidence is incremented.

When a transaction commits, various book keeping for that transaction needs to happen for accurate scheduling in the future. These operations are shown in pseudo-code in Example 4.5. These items are the average transaction size, the confidence between dTxIDs if one serialized against the other, and the average similarity of the committed transaction. To update the confidence of a conflict occurring in the future between two transactions that serialized the respective Bloom filters are intersected. If an intersection is not null then the confidence is incremented, otherwise it is decremented weighted by similarity.

Updating similarity is the most expensive part of BFGTS. As seen in pseudo-code in

Example 4.5 Pseudo code for routines used during Transaction Commit for the BFGTS algorithm.

```
1  void commitTx(int dTxID)
2  {
3   updateAvgSize(dTxID);
4   updateBloom(dTxID);
5   int waitingOn = checkWasSerialized(dTxID);
6
7   if (waitingOn != NO_TX){
8    sim = 0.5 * (simOf(dTxID) + simOf(waitingOn));
9    if(intersectBlooms(dTxID, waitingOn)) {
10    incConfProb(dTxID, waitingOn, incVal*sim);
11   } else {
12    decConfProb(dTxID, waitingOn, decVal*(1-sim));
13   }
14  }
15 }
16
17 void updateBloom(int dTxID)
18 {
19  nBloom = readCPUBloomFilter();
20  uBloom = UNION(nBloom, bloomFilterTable[dTxID]);
21  newSim = calcSim(nBloom, bloomFilterTable[dTxID], uBloom);
22  newSim = newSim/txStats[dTxID].avgTxSize;
23  txStats[dTxID].sim = 0.5*(txStats[dTxID].sim+newSim);
24 }
25
26 double calcSim(nBloom,oBloom,uBloom)
27 {
28  den = NUMHASHBITS*ln(1-1/NUMBLOOMBITS);
29  newSize = ln(1-(bitCnt(nBloom)/NUMBLOOMBITS))/den;
30  oldSize = ln(1-(bitCnt(oBloom)/NUMBLOOMBITS))/den;
31  unionSize = ln(1-(bitCnt(uBloom)/NUMBLOOMBITS))/den;
32  return (newSize+oldSize-unionSize);
33 }
```

Example 4.5, calculating similarity requires two expensive functions: `bitCnt()`, and `ln()`. However, modern ISAs support both operations at the instruction level. A low latency 64-bit wide population count instruction, and a floating point logarithm instruction exist in modern ISAs such as x86. These instructions are: `popcnt` and `fyl2x` [4]. The latencies of these instructions are 2-cycles and 13-cycles respectively for the AMD K10 architecture [57] and these latencies are assumed for the modeled system in the evaluation section of this chapter.

The transaction commit stage of the scheduling runtime can be particularly expensive, especially for small transactions, adding 100's of cycles of overhead to a transaction that may only be a few 10's of cycles in length. To reduce the overhead for small transactions similarity is updated for these transactions once every *n* commits. Large transactions are able to amortize the overhead of updating similarity on every commit, and usually benefit from the added scheduling accuracy.

4.2.3 BFGTS-HW/Backoff Algorithm

Example 4.6 Predictor Hardware setup Pseudo Code for Hybrid BFGTS predictor

```
1  void predictorSetup(int cpuID, int dTxID)
2  {
3   sTxID = dTxID >> shift_value;
4   confTableBase = &conf_table[cpuID][sTxID*num_txs];
5
6   XACTION_QUERYPRED(confTableBase, SETCONFTABLE_BASEADDR);
7
8   if (checkConflictPressure(sTxID)<=PRESSURE_THRESHOLD) {
9       statsTable[dTxID].sched_mode = BACKOFF;
10      XACTION_QUERYPRED(DISABLE_PREDICTOR,
11                        SET_PREDICTOR_OPMODE);
12  } else {
13      statsTable[dTxID].sched_mode = BFGTS;
14      XACTION_QUERYPRED(ENABLE_PREDICTOR,
15                        SET_PREDICTOR_OPMODE);
16  }
17 }
```

To further reduce overhead, we present a hybrid BFGTS predictor borrowing ideas from Yoo and Lee's [113] ATS to allow the runtime to switch between using randomized backoff when contention is low and BFGTS using the hardware accelerator when contention is high. To measure contention ATS's metric *conflict pressure* is used to determine when to switch between BFGTS and randomized backoff with the goal of saving execution overhead. To implement the HW/Backoff predictor changes are made to the presented BFGTS algorithm and described in the following paragraphs along with example pseudo-code which was omitted from [28] due to space concerns.

Just before executing the `TM_BEGIN` instruction the runtime checks the conflict pressure for the *sTxID* that wishes to execute, if it is over a set threshold then BFGTS is (by enabling the hardware) enabled and a scheduling prediction is made to suspend or continue execution. Otherwise, no prediction is made (the hardware is disabled) and the transaction begins

Example 4.7 Suspend Transaction Handling Pseudo Code for Hybrid BFGTS predictor

```
1 void suspendTx(int dTxID, int dTxIDSusp)
2 {
3  sim = 0.5 * (simOf(dTxID) + simOf(dTxIDSusp));
4  decay = decayVal * (1 - sim);
5  decConfProb(sTxID, sTxIDSusp, decay);
6  statsTable[dTxID].txWaitingOn = dTxIDSusp;
7  statsTable[sTxID].conflictPressure =
8    statsTable[sTxID].conflictPressure * ALPHA + (1.0 - ALPHA);
9  if(avgTxSize(dTxIDSusp) >= SMALL_TX_SIZE){
10   pthread_yield();
11  } else {
12   stallOnTx(dTxIDSusp);
13  }
14  restore_checkpoint();
15 }
```

execution. This allows the BFGTS-HW/Backoff predictor to save overhead on transaction begins by not always having to walk the *CPU Table* on `TX_BEGIN`. This operation is shown in Example 4.6 when the predictor hardware is being setup with the proper address for indexing the *Confidence Table*.

Example 4.8 Conflict Handling Pseudo Code for Hybrid BFGTS predictor

```
1 void txConflict(int dTxId,int dTxIdConf)
2 {
3  statsTable[sTxID].conflictPressure =
4    statsTable[sTxID].conflictPressure * ALPHA + (1.0 - ALPHA);
5
6  if (statsTable[sTxID].conflictPressure>PRESSURE_THRESHOLD) {
7    sim = 0.5 * (simOf(dTxId) + simOf(dTxIdConf));
8    inc = incVal * sim;
9    incConflictProb(dTxID, dTxIDConf, inc);
10   incConflictProb(dTxIDConf, dTxID, inc);
11   XACTION_QUERYPRED(ENABLE_PREDICTOR,
12                     SET_PREDICTOR_OPMODE);
13  } else {
14   doRandomizedLinearBackoff(dTxID);
15  }
16 }
```

To properly support switching between BFGTS and Backoff, modifications had to be made to the functions that are called when transactions conflict or when a transaction is suspended by BFGTS that involve updating the *conflict pressure*. To update *conflict*

pressure, the BFGTS-HW/Backoff algorithm increases pressure on aborts in the function `txConflict()` shown in Example 4.8, and predicted conflicts in `suspendTx()` shown in Example 4.7 in the same fashion as ATS. As shown in the examples, to update conflict pressure and turn BFGTS on and off, there are two variables that are used: `ALPHA` and `PRESSURE_THRESHOLD`. These are parameters that can be set in software for each benchmark. The `PRESSURE_THRESHOLD` is exactly like it sounds, it is a number between 0 and 1 that is used to switch between the two contention managers. The `ALPHA` value is the weight given to past history. A high `ALPHA` tends to increase or decrease the conflict pressure slowly, making the switching between BFGTS and backoff slow.

Example 4.9 Pseudo code for commit routine used during Transaction Commit for Hybrid BFGTS predictor.

```
1  void commitTx(int dTxID)
2  {
3   if (statsTable[sTxID].conflictPressure > PRESSURE_THRESHOLD ||
4       statsTable[dTxID].schedMode == BFGTS) {
5     updateAvgSize(dTxID);
6     updateBloom(dTxID);
7     int waitingOn=
8       checkWasSerialized(dTxID);
9
10    if (waitingOn!=NO_TX){
11      sim=0.5*(simOf(dTxID)+simOf(waitingOn));
12      if(intersectBlooms(dTxID,waitingOn)){
13        incConfProb(dTxID,waitingOn,incVal*sim);
14      } else {
15        decConfProb(dTxID,waitingOn,decVal*(1-sim));
16      }
17    }
18    statsTable[sTxID].conflictPressure =
19      statsTable[sTxID].conflictPressure * ALPHA;
20
21    if (statsTable[sTxID].conflictPressure<=PRESSURE_THRESHOLD) {
22       zeroBloom(dTxID);
23       statsTable[dTxID].sched_mode = BACKOFF;
24       XACTION_QUERYPRED(DISABLE_PREDICTOR,
25                                                  SET_PREDICTOR_OPMODE);
26     }
27  } else {
28    statsTable[sTxID].conflictPressure =
29      statsTable[sTxID].conflictPressure * ALPHA;
30  }
31 }
```

When a transaction commits, it checks *conflict pressure* first in `commitTx()` shown in Example 4.9 to determine if the transaction needs to perform the Bloom filter calculations and other book keeping operations such as updating the average transaction size. When *conflict pressure* is low, `commitTx()` skips performing these calculations eliminating scheduling overhead. On commits, BFGTS-HW/Backoff decreases *conflict pressure* as seen in the pseudo-code. Also note that when a switch is made between BFGTS and Backoff in `commitTx()`, that the Bloom filter for that dTxID is set to zero so conflict confidence values are not incremented for this particular transaction if other transactions serialize on it. Section 4.3 will show that being able to switch between BFGTS and randomized backoff eliminates enough overhead to allow larger Bloom filters to be used and in some cases increase performance.

4.3 Evaluation

4.3.1 Simulation Environment and Benchmarks

The M5 Full System Simulator [26] is again used to evaluate BFGTS. Like in the PTS chapter the baseline TM system is based on LogTM [84] and has Operating System (OS) support. Three different scheduling based contention managers are evaluated in this chapter: Adaptive Transaction Scheduling (ATS), Proactive Transaction Scheduling (PTS), and Bloom Filter Guided Transaction Scheduling (BFGTS). The simulation parameters are detailed in Table 4.3. The latencies for the `popcnt` and `fyl2x` instructions are modeled as well. The hardware accelerator with accompanying *Tx Confidence Cache* size as described in Table 4.3 has an area overhead of $\sim$**3%** of one 64kB L1 data cache in the system modeled and there is a hardware accelerator per processor core.

The experiments for ATS, PTS and the five BFGTS variants assume an overcommitted system with 64 threads with four threads assigned per processor. This configuration

Benchmark	Input Parameters
Delaunay [72]	-i large.2 -m30 -t64
Genome	-g4096 -s32 -n524288 -t64
Kmeans	-m20 -n20 -t0.05 -i random50000_12 -p64
Vacation	-n8 -q10 -u80 -r65536 -t131072 -c64
Intruder	-a10 -l32 -n8192 -s1 -t64
Ssca2	-s15 -i1.0 -u1.0 -l3 -p3 -t64
Labyrinth	-i random-x96-y96-z3-n128.txt -t64
Yada	-i large.2 -m30 -t64

Table 4.2: STAMP Benchmark input parameters.

Feature	Description
Processors	16 one IPC Alpha cores @ 2GHz
Special Instructions	popcnt 2-cycle latency, fyl2x 15-cycle latency
L1 Caches **Tx Confidence Cache** **L2 Cache** **Interconnect**	64kB, 1 cycle latency, 2-way associative, 64-byte line size 2kB, 16-way associative 1 cycle latency, 64-byte line size 32MB, 32 cycle latency, 16-way associative, 64-byte line size Shared bus at 2GHz
Main Memory	2048MB, 100 cycles latency
Linux Kernel	Modified v2.6.18
Contention Managers	PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff, BFGTS-HW/Backoff, BFGTS-NoOverhead
Signature Size	512bit-8192bit for BFGTS commit routines, perfect signature used for conflict detection.

Table 4.3: M5 Simulation Parameters.

was chosen because an overcommitted system is typical for systems running an OS. The advantage of such overcommitted systems is that when a thread blocks the OS can switch in another thread. This avoids leaving a core idle, thus increasing throughput. The dynamically tuning software version of ATS developed by Yoo and Lee [113] is tested using pthreads to suspend and wake threads when throttling. The full version of PTS using the best performing variants from Chapter 3 is compared against. Five versions of BFGTS: The hardware accelerated version presented in the previous sections BFGTS-HW, an all software version called BFGTS-SW, two hybrid BFGTS algorithms from Section 4.2.3 combining BFGTS-SW or BFGTS-HW with Backoff managers called BFGTS-SW/Backoff and BFGTS-HW/Backoff respectively, and BFGTS-NoOverhead. BFGTS-NoOverhead, as its name implies, implements BFGTS where all the software functions presented in Section 4.2.2.2 complete in one cycle. This is done to evaluate how well BFGTS predicts and schedules around conflicts when it does not have to amortize the cost of book keeping operations. BFGTS-NoOverhead also uses perfect read/write set signatures.

The transaction schedulers in this chapter are evaluated with the STAMP benchmark suite [37]. The benchmark parameters are shown in Table 4.2. And are identical to those tested in the Chapter 3.

This chapter presents a similar analysis to chapter 3, but there are more parameters to investigate as evident by the numerous sensitivity studies that will be presented. Therefore at the start of a section presenting results, details of the parameter settings will be included where applicable. The relevant parameters are the following: 1) Minimum, Max-

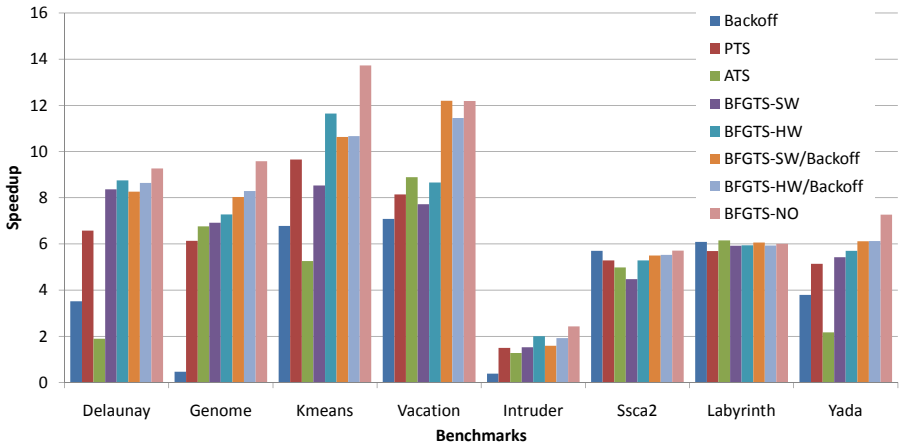


Figure 4.5: Speedup of Backoff, PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff Hybrid, BFGTS-HW/Backoff Hybrid and BFGTS-NoOverhead on a 16 processor system over 1 core

imum confidence values and conflict threshold - fixed to 0, 255 and 128 respectively. 2) Confidence increment/decrement value - fixed to increment/decrement by 50 weighted by similarity when confidence is modified. 3) Small transaction threshold - fixed to 10 cache lines (approximately double the size of queue/dequeue operations in STAMP benchmark suite). 4) Bloom filter size - variable between 512bit-8192bit, uses k=1 H3 [39] hash function. 5) Small Tx Update interval - variable. 6) Time Decay factor - variable. 7) Past history weighting of conflict pressure (Alpha) - variable. 8) Conflict pressure threshold for switching between backoff and BFGTS - variable.

4.3.2 Performance Analysis

This section presents a performance analysis of the five tested BFGTS variants: BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff. Figure 4.5 shows the overall speedup over one core for a 16 processor system for the BFGTS techniques as well as speedups for Backoff, PTS and ATS. Figure 4.6 better shows the performance differences of BFGTS over PTS as a baseline system. Table 4.4 shows a breakdown of the contention experienced for each of the tested contention managers. In this section the Bloom filters are variable, and the best performing Bloom filter is presented in the results as we assume a hardware Bloom filter to allow variable size. The “Small Tx Update interval” is fixed to 20, “Time Decay factor” is fixed to 7, “Alpha” is set to 90% and “Conflict Pressure Threshold” is fixed to 25% for this section.

Backoff is presented here again to show all other tested proactive contention managers

tested in this chapter are better by a large margin. As can be seen in Figure 4.5, backoff performs the worse on average across all the benchmarks, and in the case of *Genome* and *Intruder* has performance worse than serial due to a high amount of contention. As seen in Table 4.4, the backoff contention suffers from substantially more contention than all the other tested contention managers. Again, as in chapter 3, backoff does perform the best for the *Ssca2* benchmark because of its extremely low contention and backoff's low overhead. Backoff also performs well for the *Labyrinth* benchmark.

The PTS configuration presented here is the best performing configuration as seen in chapter 3 for each benchmark, using optimal sized Bloom filters and code optimizations. As can be seen, PTS is competitive with both BFGTS and ATS, and is therefore the baseline of comparison in Figure 4.6. For all the benchmarks, PTS is beaten by the BFGTS variants. As seen by Table 4.4, PTS does have a problem with being overly optimistic at times due to its implementation. This leads to higher contention than all the other proactive contention managers and in cases, lower performance.

ATS, as described in the previous sections is a very simple proactive contention manager using only contention rate as the indication of when to schedule or not schedule. Because of this coarse grained scheduling method, ATS suffers from being the most pessimistic of the schedulers presented here. For some benchmarks it over schedules and degrades the performance towards serial execution prematurely. This is evident from the speedups for the *Delaunay*, *Kmeans* and *Yada* benchmarks in Figure 4.5. Looking at the contention experienced by ATS for these benchmarks in Table 4.4, it is seen that it is the lowest for these benchmarks. Part of the reason for this is from the complex conflict patterns seen by these benchmarks (see Table 4.1). For benchmarks with less complex conflict patterns, but still high contention such as *Genome* and *Intruder*, ATS performs well and conflict pressure is a surprisingly good metric for basing scheduling decisions upon. Still, the overall design of ATS limits the maximum amount of parallelism that can be extracted when contention is present. Overall, ATS is 15% worse than PTS in performance and anywhere from 17%-35% worse in performance to BFGTS (excluding BFGTS-NO) as seen in Figure 4.6.

BFGTS-SW, as seen in Figures 4.5 and 4.6 has mixed performance when compared to PTS and ATS. For benchmarks with high contention, and transactions that can tolerate an all software implementation of BFGTS, it can improve performance by a fair amount as seen in Figure 4.6, getting a maximum of 27% better performance than PTS in the *Delaunay* benchmark. In benchmarks like *Kmeans* and *Ssca2* that have small transactions, the overhead leads to worse performance than PTS as BFGTS does not use any application specific optimizations, and must rely on more complicated Bloom filter manipulations. These

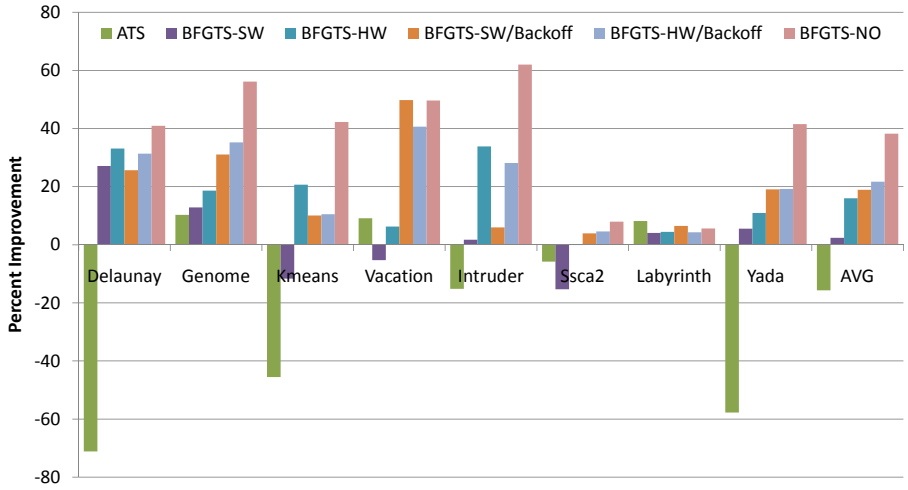


Figure 4.6: Percent difference of ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff Hybrid, BFGTS-HW/Backoff Hybrid and BFGTS-NoOverhead over PTS on a 16 processor system over 1 core

penalties lead to an only 2% performance improvement over PTS as seen in Figure 4.6.

BFGTS-HW, as seen in Figures 4.5 and 4.6 benefits greatly from the addition of the hardware accelerator for performing the scheduling operations before a `TX_BEGIN`. This leads to BFGTS-HW performing better than, or equivalent to PTS and ATS for all benchmarks, including benchmarks like *Kmeans* and *Ssca2* where overhead is extremely important to minimize. Even so, overhead remains in BFGTS-HW as it still performs scheduling operations and Bloom filter manipulations. Overall BFGTS-HW gets a 16% performance improvement over PTS.

BFGTS-SW/Backoff combines BFGTS-SW with the ATS derived switching method presented in Section 4.2.3. This helps eliminate many of the overheads present in the BFGTS-SW, and boosts performance well above that achieved by BFGTS-SW as seen in Figures 4.5 and 4.6. This is due to turning on BFGTS only when needed. The *Ssca2*, *Vacation*, *Yada* and *Genome* benchmarks see significant improvement by turning BFGTS on and off. For *Ssca2*, due to the low contention and small transactions, BFGTS is always off and only some overhead is paid for checking the contention pressure. For *Vacation*, large gains are seen because the benchmark constructs a red-black tree during execution. As the tree is constructed, eventually the transactions are operating in different parts of the tree and BFGTS can be turned off and backoff can be used. The reason BFGTS-SW and BFGTS-HW do not perform as well here is because as the tree grows, the transaction size increases. This leads to the Bloom filters falsely indicating future conflicts, and the confidence does not decrease quickly. For the *Genome* benchmark, the first part of the benchmark consists

	Backoff	PTS	ATS	BFGTS-SW	BFGTS-HW	BFGTS-SW/ Backoff	BFGTS-HW/ Backoff
Delaunay	73.4%	26.6%	23.9%	22.2%	21.6%	23.4%	23.6%
Genome	49.7%	1.8%	1.0%	1.0%	1.1%	4.1%	3.2%
Kmeans	20.6%	3.7%	0.7%	1.1%	1.8%	6.4%	6.5%
Vacation	10.3%	8.5%	3.2%	4.6%	3.1%	4.2%	5.3%
Intruder	70.1%	7.5%	4.0%	5.4%	3.2%	12.3%	9.8%
Ssca2	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
Labyrinth	18.2%	21.5%	6.4%	12.1%	9.3%	10.4%	9.3%
Yada	54.6%	27.8%	10.5%	13.7%	14.7%	16.6%	14.7%

Table 4.4: Contention experienced for each contention management technique: Backoff, PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff Hybrid and BFGTS-HW/Backoff Hybrid for a 16 processor system.

of small transactions that are inserting into a hash table with no contention. For the *Yada* benchmark, it appears to benefit primarily from the increased optimism gained from turning BFGTS on and off because it experiences more contention from Table 4.4, but gets better performance as seen in Figure 4.5. For the remaining benchmarks, BFGTS-SW/Backoff gets slightly worse performance than BFGTS-HW due to it increasing contention from cycling between BFGTS and Backoff throughout the execution of the benchmark. Overall BFGTS-SW/Backoff gets on average 18% better performance than PTS.

BFGTS-HW/Backoff adds the benefit of the hardware accelerator to reduce the overhead as well as enabling switching between BFGTS and Backoff. As can be seen it extracts even more performance than BFGTS-SW/Backoff on average, getting 21% better performance overall than PTS. It attains its performance advantages for the same reason as BFGTS-SW/Backoff, and performs similarly for all benchmarks except for the *Intruder* benchmark. For *Intruder*, the addition of the hardware accelerator eliminates overhead experienced by BFGTS-SW/Backoff, and is where the extra performance is coming from. As with BFGTS-SW/Backoff, BFGTS-HW/Backoff performs slightly worse than BFGTS-HW for some benchmarks due to the increased contention as seen in Table 4.4.

A No Overhead version of BFGTS called BFGTS-NO was also tested to demonstrate the potential of the BFGTS scheduling algorithm. It performs all the BFGTS operations instantaneously inside of M5 and uses perfect signatures that have no aliasing unlike Bloom filters (this is not a realizable implementation). As can be seen from Figures 4.5 and 4.6, it attains the best performance over all the contention managers tested. It gets on average 38% better performance over PTS. The BFGTS variants tested with realistic implementations get within 50% of the No Overhead versions performance, showing that there is more

performance left to be gained.

4.3.3 Execution Time Breakdown

This section provides analysis of where the PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff algorithms spend execution time and a detailed accounting of the overheads incurred. Figure 4.7 shows the overall runtime normalized to a single processor execution and how that time is spent. Figure 4.8 shows the proportion of time spent in each type of execution time category. These categories are the following: 1) Non-Trans - Time spent in user mode that is not inside a transaction or related to scheduling a transaction, 2) Kernel - Time spent operating in the Linux kernel, 3) Trans - Time spent executing in a transaction, this counts both committed and aborted transactions, 4) Abort - Time spent rolling back a transaction's write set, 5) Escape - Time spent suspending a transaction to service PAL code operations, like filling a TLB miss, 6) Sched Begin - Time spent executing the `scheduleTx()` and `suspendTx()` function of BFGTS, 7) Sched Abort - Time spent executing in the `txConflict()` function, 8) Sched Commit - Time spent executing in the `commitTx()` function.

As can be seen in Figures 4.7 and 4.8, most of the benchmarks except for *Kmeans* and *Ssca2* spend the majority of time executing transactional code, and because of the proactive nature of the schedulers tested, very little time is spent in "Abort" and re-executing code as seen by the small differences in the size of the "Transactional" sections of the bars, as well as from the small differences in contention as shown in Table 4.4. The main differences in the different contention managers is the time spent in scheduling operations that enact serialization on the transactions executing in different threads.

For PTS, the main differences between this proactive manager and the BFGTS technique is the time spent in scheduling functions. For PTS on average it spends more time in "Schedule Begin" than any of the BFGTS techniques as seen in Figures 4.7 and 4.8. This means that it is spending a fair amount of time serializing execution and scanning the *CPU Array* to determine if scheduling can proceed, lending support to the decision to use hardware acceleration. This amount of time spent in "Sched Begin" is also from the rudimentary Bloom filter operations, leading to more predictions of conflicts being likely. This rudimentary Bloom filter use is supported by the small amount of time spent in "Sched Commit".

As seen in the previous section ATS suffers from over serialization due to its design using a centralized wait queue to enact scheduling and using coarse grained information about instantaneous contention. The over serialization shows up in the "Kernel" section of the bars in Figures 4.7 and 4.8, where the OS is putting threads on wait queues and taking

them off when the threads are woken. This large amount of scheduling overhead is evident in the *Delaunay*, *Kmeans*, *Intruder* and *Yada* benchmarks where the ATS scheduler spends a large percentage of its time in the OS. For benchmarks where ATS performs well, such as the *Genome*, *Vacation* and *Ssca2*, ATS has almost no overhead present in "Sched Begin", "Sched Abort", or "Sched Commit" due to its very minimalist implementation.

For the BFGTS variants, they are spending less time in the scheduling subroutines on average than PTS or ATS as seen in Figures 4.7 and 4.8. This is due to BFGTS finding more parallelism by using the Bloom filter manipulations covered in the previous sections. These Bloom filter manipulations are expensive though. For example, in the *Genome* benchmark, BFGTS-HW decreases the amount of time spent in "Sched Begin" significantly compared to PTS. This advantage is less than it could be because BFGTS-HW adds some overhead back to the execution in the form of more "Sched Commit" operations compared to PTS as seen by the larger bar. For the BFGTS-SW/Backoff and BFGTS-HW/Backoff techniques, they eliminate much of this overhead by selectively turning BFGTS on and off, and therefore see a large performance gain for the *Genome* benchmark. For the rest of the benchmarks, this trend holds true, the BFGTS variants use on average less scheduling overhead due to them finding more parallelism on average. For a benchmark like *Ssca2* where overhead incurs a large penalty due to its small amount time spent in transactions, all the proactive techniques incur some overhead that is unavoidable as they still have to check if scheduling is required.

Table 4.5 shows the overhead in cycles per transaction commit incurred by the BFGTS-SW and BFGTS-HW techniques in the same fashion as the overhead tables presented in Chapter 3. As can be seen in the table, BFGTS-SW and BFGTS-HW incur similar amounts of overhead for benchmarks that have on average large transactions ($>$1000 cycles) and can amortize the cost of scheduling by leveraging more parallelism. For benchmarks with small transactions on average, BFGTS-SW and BFGTS-HW incur a large amount of overhead, in some cases up to 25x the size of the transaction itself in the case of *Ssca2* for BFGTS-SW. The hardware accelerator used in BFGTS-HW does make a large difference for benchmarks with small transactions. For example the *Kmeans* benchmark has a overhead of 12x for BFGTS-SW, but the hardware accelerator in BFGTS-HW cuts this down to 3.5x, which while still high, is a large decrease in overhead. Overall, the range of overhead is from 45%-2523% for BFGTS-SW and BFGTS-HW. This range is almost identical to PTS from chapter 3. But, there are significant differences between specific benchmarks. For example, the *Delaunay* benchmarks sees a 105% overhead with PTS, with most of it in the "Sched Begin" phase of scheduling. BFGTS, on the other hand only sees a 49% overhead for BFGTS-HW with substantially less cycles spent in all phases of scheduling. This indicates

	Delaunay BFGTS		Genome BFGTS		Kmeans BFGTS		Vacation BFGTS	
	-SW	-HW	-SW	-HW	-SW	HW	-SW	-HW
Transactional	1269	1262	353	353	12	12	683	672
Kernel	24	22	247	245	13	13	228	187
Sched Begin	537	452	180	140	96	11	240	161
Sched Abort	10	11	0	0	0	0	1	1
Sched Commit	143	137	134	126	41	28	172	156
Total	714	622	561	511	150	52	641	505
w/o Kernel	690	600	314	266	137	39	413	318
Pct Overhead	56%	49%	159%	145%	1250%	347%	94%	75%
w/o Kernel	54%	48%	89%	75%	1142%	260%	60%	47%
	Intruder BFGTS		Ssca2 BFGTS		Labyrinth BFGTS		Yada BFGTS	
	-SW	-HW	-SW	-HW	-SW	-HW	-SW	-HW
Transactional	76	68	13	13	740562	747266	1028	1045
Kernel	16	11	265	240	295665	284831	21	19
Sched Begin	638	470	41	0	34558	34996	1126	1045
Sched Abort	1	1	0	0	12	3	5	6
Sched Commit	136	92	22	17	96	248	180	78
Total	791	574	328	257	330331	320078	1332	1148
w/o Kernel	775	563	63	17	34666	35247	1311	1129
Pct Overhead	1041%	844%	2523%	1977%	45%	43%	130%	110%
w/o Kernel	1020%	828%	485%	131%	5%	5%	128%	108%

Table 4.5: Amount of scheduling overhead experienced in cycles per transaction commit for BFGTS-SW, and BFGTS-HW.

BFGTS-HW is making better decisions overall than PTS, which is in turn leading to better performance.

Table 4.6 shows the overhead in cycles per transaction for the BFGTS-SW/Backoff and BFGTS-HW/Backoff schedulers. As seen in this table, the BFGTS/Backoff schedulers can significantly reduce overhead and therefore lead to better performance. This is seen especially in the *Vacation* benchmark, reducing overhead from 75% in BFGTS-HW to only 29% for BFGTS-HW/Backoff. Similar reductions with performance increases are seen for the *Genome*, *Yada* and *Ssca2* benchmarks as well. As seen in both overhead tables, the *Ssca2* benchmark has a large amount of kernel overhead, this is for the same reason as with PTS, due to load imbalance and poor caching properties. While these overheads are still high, there is a limit that the overhead can be reduced by, as these overheads account for time spent serializing. For benchmarks like *Intruder*, this serialization overhead is required to maintain forward progress and acceptable performance.

	Delaunay BFGTS-		Genome BFGTS-		Kmeans BFGTS-		Vacation BFGTS-	
	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off
Transactional	1271	1268	374	368	15	16	661	662
Kernel	26	36	227	195	14	14	80	110
Sched Begin	504	444	106	105	35	24	34	35
Sched Abort	15	15	2	2	2	2	2	2
Sched Commit	183	120	56	70	9	11	35	46
Total	728	615	391	372	60	51	151	193
w/o Kernel	702	579	164	177	46	37	71	83
Pct Overhead	57%	49%	105%	101%	400%	319%	23%	29%
w/o Kernel	55%	46%	44%	48%	307%	231%	11%	13%
	Intruder BFGTS-		Ssca2 BFGTS-		Labyrinth BFGTS-		Yada BFGTS-	
	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off	SW/ Back-off	HW/ Back-off
Transactional	88	77	12	12	797531	790228	1154	1141
Kernel	10	10	234	235	200973	232360	27	27
Sched Begin	597	499	6	0	38134	37728	842	955
Sched Abort	5	4	0	0	58	41	8	7
Sched Commit	92	62	3	2	39	36	198	36
Total	702	575	243	237	239204	270165	1075	1025
w/o Kernel	692	565	9	2	38231	37805	1048	998
Pct Overhead	798%	747%	2025%	1975%	30%	34%	93%	90%
w/o Kernel	786%	734%	75%	17%	5%	5%	91%	87%

Table 4.6: Amount of scheduling overhead experienced in cycles per transaction commit for BFGTS-SW/Backoff Hybrid, and BFGTS-HW/Backoff Hybrid.

4.3.4 BFGTS Sensitivity Tests

The BFGTS techniques have a larger number of variable parameters that need to be tested for their affect on performance. These include sensitivity to Bloom filter size, Small Tx Update interval, Time Decay size and ATS parameters for the BFGTS/Backoff schedulers. The following sections will provide detailed sensitivity analysis.

4.3.4.1 Bloom Filter Sensitivity

The entire BFGTS technique heavily depends on the accuracy of the Bloom filters used and the overhead they impose when involved in the *Similarity* calculations described in this

chapter. This section varies the Bloom filter size from 512bit-8192bit for the BFGTS-SW, BFGTS-HW, BFGTS-SW/Backof and BFGTS-HW/Backoff schedulers. The other variable parameters are set as follows: The "Small Tx Update interval" is fixed to 20, "Time Decay factor" is fixed to 7, "Alpha" is set to 90% and "Conflict Pressure Threshold" is fixed to 25% for this section. All other parameters covered in Section 4.3.1 are fixed to the values described there.

Figure 4.9 shows the speedup attained for each benchmark using different sized Bloom filters for BFGTS-SW. As can be seen from the figure, the general trend is that there is no best sized Bloom filter for all the benchmarks. Some prefer the smallest sized Bloom filters to keep the overhead of calculating similarity to a minimum while other benchmarks want a larger Bloom filter to attain better prediction accuracy as shown by an increase in speedup. The *Kmeans* benchmark for example performs best when the Bloom filter is sized to the smallest value of 512bits and increasing the Bloom filter size decreases the performance greatly. On the other hand, the *Vacation* benchmark prefers a larger Bloom filter of 2048bits. *Vacation* has very poor performance with a small Bloom filter due to aliasing effects of inserting many addresses into the filter causing BFGTS to predict pessimistically due to *Vacation's* large RWSet on average. Increasing the filter size beyond 2048bits though shows the overhead of doing the similarity calculations dominates runtime eventually. This trend of preferring a slightly larger Bloom filter is evident for all the benchmarks except for the *Labyrinth* benchmark that shows no clear trend. This is most likely due to this benchmark having a small number of transactions and BFGTS has limited time to learn the conflict pattern.

Figure 4.10 shows the speedup attained for the benchmarks using different Bloom filter sizes for the BFGTS-HW scheduler. As can be seen, the trends are very similar for BFGTS-SW, but the performance is better due to the overhead reduction from using the hardware accelerator on transaction begins. Some benchmarks change their preference to Bloom filter size slightly. The *Intruder* benchmark prefers a 1024bit Bloom filter for BFGTS-SW, but favors a 512bit Bloom filter when BFGTS-HW is used. This is due to the slightly more parallelism exposed by using a larger Bloom filter in BFGTS-SW that amortizes the slight increase in overhead. While for BFGTS-HW, since the overhead of scheduling is more dominant, and therefore leads to a smaller Bloom filter being optimal. As seen in Figure 4.6 from Section 4.3.2, a no-overhead version of BFGTS with perfect filters can almost double the performance realized for the *Intruder* benchmark. This is also true for the *Yada* benchmark as seen in Figure 4.10.

Figures 4.11 and 4.12 show the Bloom filter size sensitivity for the BFGTS-SW/Backoff and BFGTS-HW/Backoff schedulers respectively. The behavior of the benchmarks to the

Bloom filter size is substantially different than to the behavior shown by BFGTS-HW and BFGTS-SW. For BFGTS-SW/Backoff and BFGTS-HW/Backoff the benchmarks are much less sensitive to Bloom filter size than before. The *Genome* benchmark was sensitive to Bloom filter size when run using either BFGTS-SW or BFGTS-HW. With BFGTS-SW/Backoff and BFGTS-HW/Backoff, the bars are almost equal over the range of Bloom filter sizes. This from the substantial reduction in overhead coming from switching between BFGTS and backoff effectively. Even the *Kmeans* benchmark shows less sensitivity to Bloom filter size, being able to use Bloom filters up to 2048bit before experiencing a drop-off in performance. The *Vacation* benchmark shows the most interesting sensitivity for these schedulers. Its performance is relatively flat until a Bloom filter size of 8192bits where performance increases significantly. This happens because the *Vacation* benchmark has large RWSets as the Red-Black tree it builds grows in size as the benchmark executes, and only when the Bloom filter is sized to 8192bits can the RWSet be represented accurately to effect better predictions. The large reduction in overhead caused by switching between BFGTS and backoff allows this size Bloom filter to be used effectively for this particular benchmark. The rest of the benchmarks show relatively little sensitivity to Bloom filter size for BFGTS-SW/Backoff and BFGTS-HW/Backoff.

As seen in this study, sizing the Bloom filter properly is affected by two key variables that interact: overhead and prediction accuracy. Depending on how dominant either component is in the execution of the benchmark affects what size of Bloom filter is optimal. This raises several questions this thesis leaves as future work if a variable sized Bloom filter is assumed for the TM implementation: Can Bloom filter size be determined statically? Should the Bloom filter be sized dynamically at runtime to balance overhead and prediction accuracy?

4.3.4.2 Small Tx Update Interval

The "Small Tx Update Interval" is another parameter that warrants a sensitivity study as it affects both the overhead incurred by BFGTS and also the prediction accuracy as it eliminates similarity calculations done for small transactions (which are defined throughout this thesis as transactions touching 10 cache lines or less). This section presents results for only the BFGTS-SW and BFGTS-HW schedulers as they are the more sensitive to overheads as seen in the previous sections. The other variable parameters are set as follows: The "Bloom filter size" is the best performing size, "Time Decay factor" is fixed to 7, "Alpha" is set to 90% and "Conflict Pressure Threshold" is fixed to 25% for this section. All other parameters are fixed as described in Section 4.3.1.

Figures 4.13 and 4.14 show the sensitivity of each individual benchmark to only updat-

Interval	Avg % Improvement BFGTS-SW	BFGTS-HW
0	1.1%	12.3%
10	2.2%	15.0%
20	2.4%	15.9%
40	3.6%	15.2%
80	3.7%	15.4%
160	3.5%	15.2%

Table 4.7: Average percent improvement over PTS for BFGTS-SW and BFGTS-HW for Small Transaction Update intervals of 0, 10, 20, 40, 80, and 160 for a 16 processor system.

ing similarity for small transactions every 0,10,20,40,80 and 160 executions of the transaction. As can be seen, the overhead of updating small transactions every execution incurs very high overhead, and is the worst performer for all the benchmarks tested. As the update interval grows, this affects prediction accuracy to a small degree for benchmarks like *Vacation* and *Intruder* that show worse performance when the update interval is large for BFGTS-HW. For BFGTS-SW, the overhead is already large, so this parameter affects the performance less for individual benchmarks.

Another way to look at the effect of this parameter is to look at how it affects the overall performance gain across all the benchmarks tested. This is shown in Table 4.7 as percentage improvement over PTS. As can be seen, updating all transactions every execution has the worst performance for both BFGTS-HW and BFGTS-SW. As the update interval is increased, average performance improvement gradually increases. For BFGTS-SW the update interval with the best performance is 80 commits before updating similarity for small transactions at 3.7% better than PTS. After 80 the performance levels off, and gets slightly worse. For BFGTS-HW, best interval is 20 commits before updating getting almost 16% better performance on average, larger intervals see performance level off at a slightly worse average improvement. This study shows once again that there is a balance between cutting overhead and sacrificing prediction accuracy as seen in the Bloom filter sensitivity study.

4.3.4.3 Time Decay Factor

The "Time Decay Factor" as described in Section 4.2.2 is hysteresis applied to confidence values indicating conflict between transactions. As a transaction predicts conflict, it will slightly decrease confidence in future conflicts occurring to combat staleness in confidence values leading to pessimistic predictions. In this section I vary the amount by which decay can decrease confidence if there is zero similarity present in a transaction (com-

pletely disjoint accesses, indicating completely transient conflicts) from 0-50. The other parameters are set the same as they were in Section 4.3.4.2.

Figure 4.15 shows the performance of each benchmark for the BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-SW/Backoff schedulers as the time decay is varied. Except for *Labyrinth* and *Ssca2*, all the benchmarks are highly sensitive to varying the decay applied to confidence values.

The *Kmeans* and *Intruder* show some sensitivity to decay. For small decay values between 0 and 10 they showcase little variance in performance. This correlates with Table 4.1 where the transactions show a relatively high degree of similarity for the transactions present in these benchmarks. This means that decay has little affect on the confidence values when it is a small value, maintaining accurate predictions. When decay becomes large, it has a large affect on confidence values and causes overly optimistic predictions, which in turn degrade performance greatly for all the scheduling techniques.

The *Genome* and *Vacation* shown more interesting behavior. They also show a high sensitivity to decay, but both when decay is small and when it is large. When decay is small, the performance of the BFGTS technique on these benchmarks is poor because BFGTS makes overly pessimistic predictions due to the connectivity of the their conflict graph along with medium similarity showing some tendency towards transient conflicts. In these cases a moderate amount of decay is best, between 5 and 10 as shown in Figure 4.15. When decay becomes too large, these benchmarks also suffer from overly optimistic predictions that severely degrade performance.

The final two benchmarks *Delaunay* and *Yada* show very different behavior. They show poor performance when decay is small, but performance steadily increases as decay grows large—the best performance is when decay is set to 50, by far the worst performing decay value for all the other benchmarks. This is due to both of these benchmarks having highly connected conflict graphs, where the transaction with the most conflicts also has a very low similarity. This means conflicts, while frequent, are also very transient and confidence needs to decay quickly to avoid stale confidence values.

As seen in this section, decay showcases very interesting properties in all the benchmarks, and is a parameter that can greatly change the performance of all the BFGTS techniques. From these results, decay is best set to a moderate value to get acceptable performance for all benchmarks.

4.3.4.4 Switch Threshold and Past History Weighting

The BFGTS-SW/Backoff and BFGTS-HW/Backoff schedulers have two parameters that can be varied that affects performance: Switch Threshold (Threshold), and Past His-

tory Weighting (Alpha). As described in Section 4.2.3 they control how quickly the scheduler switches between using BFGTS and backoff. As the threshold is increased, contention can remain higher before causing a switch between the two managers. Alpha regulates how quickly the system changes what it perceives as the contention currently present in the system. For these sensitivity studies I vary the threshold between 10% and 50% contention pressure and vary alpha between 50% (fast switching) and 95% (slow switching) for weighting the past history. The other parameters are set the same as they were in Section 4.3.4.2.

Figure 4.16 shows the results for the benchmarks as the two parameters are varied. The solid blue line shows the varying of the threshold parameter while keeping Alpha constant. The dotted red line shows the performance trend as the Alpha variable is varied.

The results are similar in nature to the sensitivity study looking at varying the decay parameter. Some applications are sensitive, and others not so sensitive. The *Ssca2* benchmark is expectedly insensitive to both parameters as its low contention keeps BFGTS switched off for the entirety of the benchmark's execution. The *Labyrinth* benchmark also appears very insensitive to either of the parameters for the BFGTS/Backoff hybrid techniques. This is likely due to the *Labyrinth* benchmark being extremely hard to predict a schedule for. This pattern matches with the previous sections that showed *Labyrinth* to be insensitive to "Decay" and even scheduler choice, all performed equally well, including randomized backoff.

The *Delaunay*, *Genome* and *Intruder* benchmarks are mildly sensitive to varying the Alpha parameter. *Delaunay* shows for BFGTS-HW/Backoff a small increase in performance as the Alpha parameter grows, indicating it prefers to switch between BFGTS and Backoff somewhat slowly (Alpha=75%). The same behavior is also seen for the *Intruder* benchmark (Alpha=90%). *Genome*, on the other hand shows the opposite trend, it prefers an Alpha that switches somewhat quickly between BFGTS and Backoff (Alpha=50%). Conversely, these same benchmarks show high sensitivity to the threshold parameter. All three of the benchmarks show degraded performance when the Threshold value is set high at 50%. This makes sense due to these benchmarks normally having high contention when using just randomized backoff, they benefit from having BFGTS switched on the majority of the time and prefer a low Threshold value. The *Genome* benchmark does see some performance degradation when the Threshold value is too low at 10% and the Alpha value is set at $>$50% for some configurations. This is due to the fact that *Genome* has some transactions that do not require BFGTS to be enabled for long periods of time due to the extra overheads incurred by using BFGTS.

The *Kmeans* and *Vacation* benchmarks show a large sensitivity to varying the Alpha pa-

rameter. For the *Kmeans* benchmark, the max performance for both BFGTS-SW/Backoff and BFGTS-HW/Backoff is attained using an Alpha value set to 75%. This makes sense as the *Kmeans* benchmark is dominated by small transactions that are negatively affected by overhead incurred when BFGTS is enabled. Still, *Kmeans* does have a large amount of contention when using randomized backoff, and therefore using BFGTS during execution is required to attain high performance. The *Vacation* benchmark behaves similarly to *Kmeans* in terms of the Alpha parameter. In this case, *Vacation* prefers a larger Alpha of 90% for BFGTS-SW/Backoff and BFGTS-HW/Backoff sees the best performance with an Alpha of 75% with a small performance decrease at 90%. *Vacation* has a large amount of contention early in the benchmark, but as the execution progresses, it sees less and less contention due to its central tree data structure getting larger. Having Alpha set larger allows BFGTS-SW/Backoff to return to backoff based contention management after the tree has grown sufficiently large. BFGTS-HW/Backoff prefers a lower Alpha due to it making more predictions over BFGTS-SW/Backoff because the overhead of transaction begin is much lower with the accelerator and keeping the conflict pressure higher, longer. Sensitivity for both benchmarks to the Threshold parameter is similar to the *Delaunay*, *Genome*, and *Intruder* benchmarks. Setting Threshold to 50% shows a precipitous drop in performance for both benchmarks due to allowing contention to get too high. *Vacation* does show different behavior when Threshold is set low at 10%, and sees less performance as it prefers the middle setting of 25% in most cases. This is due to the gains made by having less overhead from turning BFGTS off earlier than when it is set to be more optimistic at a higher value.

The *Yada* benchmarks shows almost opposite behavior to the previous benchmarks. It is insensitive to both Alpha and Threshold values when Alpha is set above 50%. It gets the best performance when Alpha and Threshold are set to 50% for BFGTS-SW/Backoff and BFGTS-HW/Backoff. This is the only benchmark to exhibit this behavior as all other benchmarks see varying degrees of performance degradation when Alpha and Threshold are set to their most optimistic settings tested in this chapter. This is behavior is due to *Yada's* complex and dense conflict graph, high contention and varying degrees of similarity that tend to be low as seen in Table 4.1. This leads to *Yada* benefitting from optimistic, yet controlled predictions using BFGTS as seen here.

As seen in this section, varying the parameters that govern how BFGTS-SW/Backoff and BFGTS-HW/Backoff switch between the high overhead BFGTS and low overhead Backoff have a sizable effect on performance. Again, each benchmark has its preferred setting of these parameters. This strengthens the design decisions made that keep BFGTS and other contention management techniques as primarily software constructs as the optimal settings are benchmark and most likely input dependent. I leave as future work methods

Percent Misspredict	Delaunay	Genome	Kmeans	Vacation
Total	42.1%	28.8%	7.3%	19.1%
Parallel	13.0%	2.6%	1.6%	5.4%
Serial	29.1%	26.1%	5.7%	13.7%
Percent Misspredict	Intruder	Ssca2	Labyrinth	Yada
Total	28.0%	0.2%	9.4%	38.1%
Parallel	5.2%	0.0%	1.2%	11.0%
Serial	22.9%	0.2%	8.2%	27.1%

Table 4.8: The BFGTS technique's prediction accuracy, as measured from the point-of-view of the algorithm.

that can potentially dynamically vary the Alpha and Threshold values to learn what the optimal values should be per benchmark and input set.

4.3.5 BFGTS Predictor Compared to PTS Predictor

As seen in this chapter, the design of the BFGTS predictor appears to be substantially better than its predecessor PTS when gauged by looking at performance over all the benchmarks tested. Still, in this section I present a qualitative evaluation of the two algorithms in terms of prediction accuracy from the point-of-view of the algorithm. These statistics were collected in the same fashion as was done for doing this evaluation for PTS. BFGTS was implemented as a zero-cycle overhead module for the M5 simulator and then statistics were gathered to characterize how often BFGTS believed it was over-serializing or predicting too optimistically.

Table 4.8 shows the inferred predictor accuracy of the BFGTS technique. The main takeaway that can be gleaned from this table when compared to Table 3.7 from Chapter 3 is the improvement in predictor accuracy. Across all benchmarks the improvement is small in most cases, but this small improvement does lead to the large improvements seen as even a small decrease in serialization can lead to large gains in parallel utilization. In general BFGTS is less likely to be overly optimistic. This is due to the fact it uses aliased TxIDs to index into the confidence tables and does not use any application specific optimizations. All these factors lead to less optimistic predictions, but the use of "Similarity" allows BFGTS to make up for these design decisions. BFGTS also sees a small increase in predicting to serialize when it should not, but it also has many benchmarks where the opposite is true in the cases of *Genome*, *Vacation* and *Intruder* which is also due to using "Similarity" to accurately separate the transient conflicts from the persistent conflicts. In total, BFGTS

is much better at predicting future transaction conflicts and picking a better schedule of transactions over PTS as seen here.

4.3.6 Potential Corner Cases for BFGTS

As with any heuristic there are corner case conditions under which the hueristic could theoretically perform poorly. The BFGTS hueristic is no different. This section covers the potential corner case.

A corner case condition that may cause BFGTS to perform poorly is the following condition: Two transactions touch varying locations in memory and are of varying size over their execution history, leading to low historical similarity, but with a high probability conflict on exactly one memory location that can be either fixed or move along with the transaction. Because of the low similarity the BFGTS hueristic may tend to treat these conflicts as transient when in fact they are not. While this would seem to be a major limiter, such a condition was not seen in practice. Even if this case did exist, BFGTS would still handle these transactions properly and not degenerate to a pathological reactive behavior due to the following:

1. When a conflict is seen, the transactions will serialize due to knowledge that the conflict exists with 100% confidence.

2. When the serializing transaction commits it will compare its RWSet to the other transaction's RWSet and find that the serializing was useful, causing confidence to be increased.

3. If decay is set to a sane value, it will not decay the edge faster than increases to confidence caused by subsequent commits recognizing the conflict still exists and serialization is useful.

Therefore, because of these reasons, this type of behavior would perform sub-optimally under BFGTS but not cause pathological behavior. This type of behavior could be found in operations like inserts into hash tables where a size variable is incremented after the inserts are completed. Similarity would be low due to the inherent randomness of hash table inserts but a conflict would always be present. In this case BFGTS would not be optimal, but orthogonal techniques meant to deal with these ancillary updates (e.g. RetCon [34]) could be applied.

4.4 Conclusions

This chapter presented the "Bloom Filter Guided Transaction Scheduler" (BFGTS) method for managing contention in an Eager/Eager HTM. The main goal of this chapter was to show the design and implementation of a better scheduling contention manager that both improved the performance of the PTS scheduling manager presented in the previous chapter that did not require application specific optimizations for performance. Both these goals were attained through the use of simple hardware to accelerate operations done on transaction begin, and more importantly through the use of the "Similarity" metric.

Similarity is the key observation that makes the BFGTS contention manager achieve better overall performance while using less resources and application knowledge than PTS. By measuring a transactions "Similarity", very specifically its self-similarity that gives a characterization of a transaction's historical RWSet, the BFGTS contention manager is able to infer the degree to which conflicts can be transient or persistent. This can be used to throttle confidence updates to get a better degree of optimism or pessimism to future conflict predictions. Similarity is also used to allow for decay to be applied to confidence to account for "staleness" in confidence values which could not be done accurately in PTS.

As shown by the evaluation section in this chapter, BFGTS can get up to 21% performance improvement over PTS on average. Compared to a competing scheduler ATS, BFGTS does even better, getting over 30% improvement even though BFGTS has higher overhead than ATS.

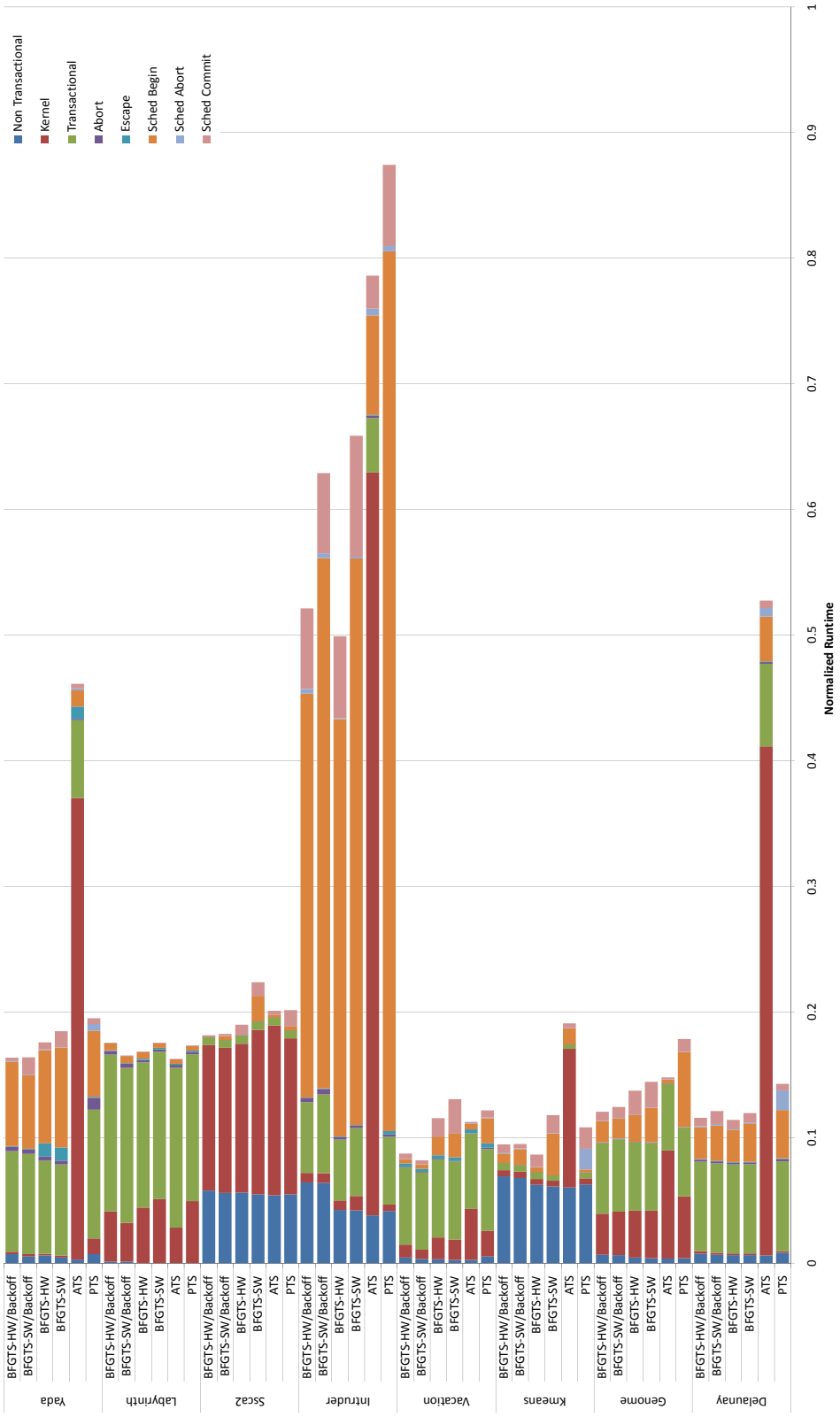


Figure 4.7: Breakdown of where time is spent for PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff Hybrid, and BFGTS-HW/Backoff Hybrid

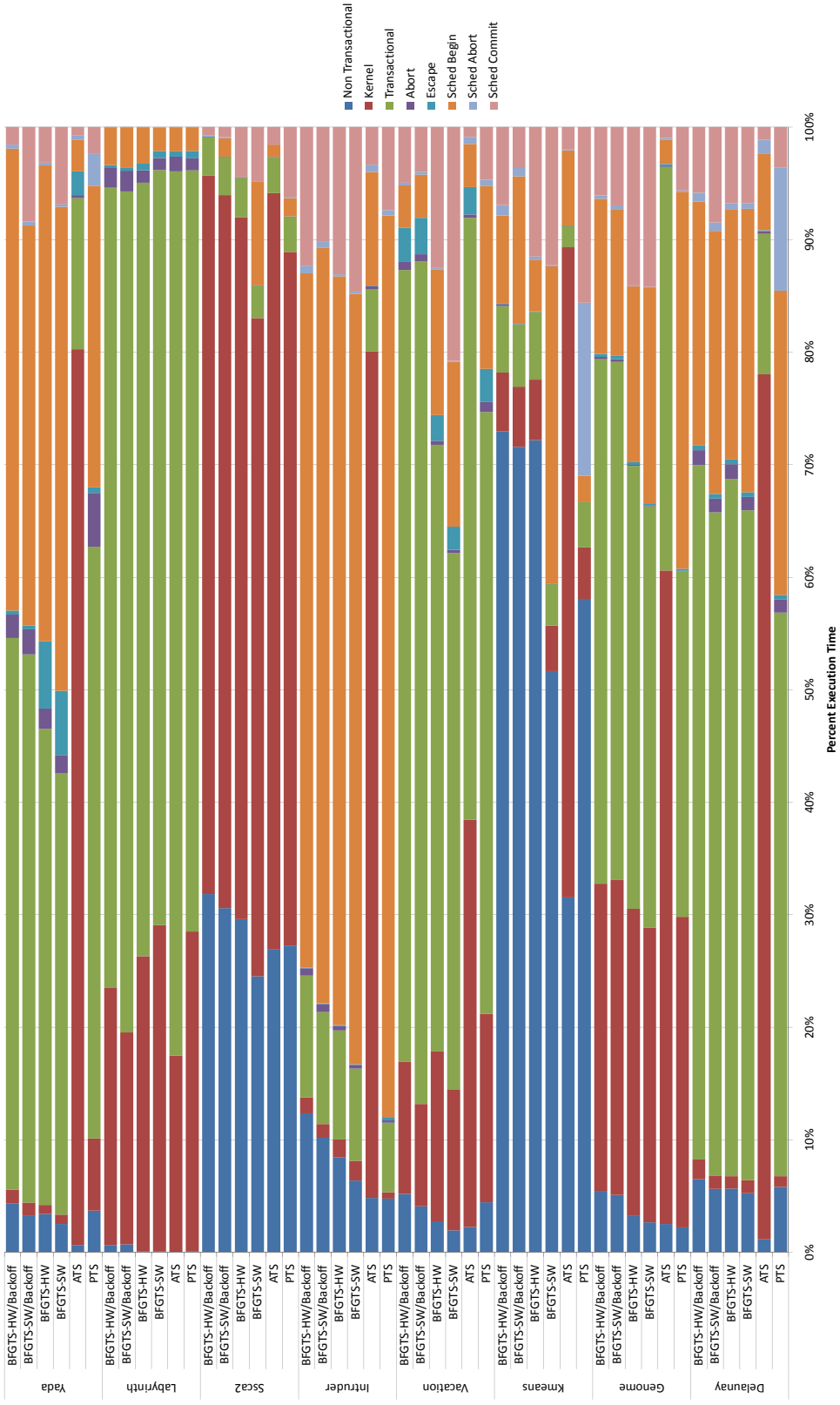


Figure 4.8: Distribution of where time is spent in the PTS, ATS, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff Hybrid, and BFGTS-HW/Backoff predictors, each benchmark is normalized to its own runtime.

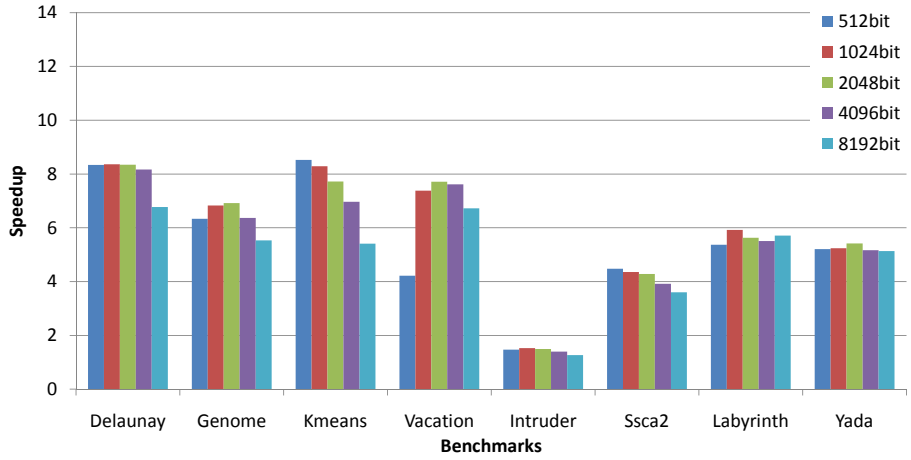


Figure 4.9: Sensitivity of BFGTS-SW to Bloom filter sizes ranging from 512bit-8192bit.

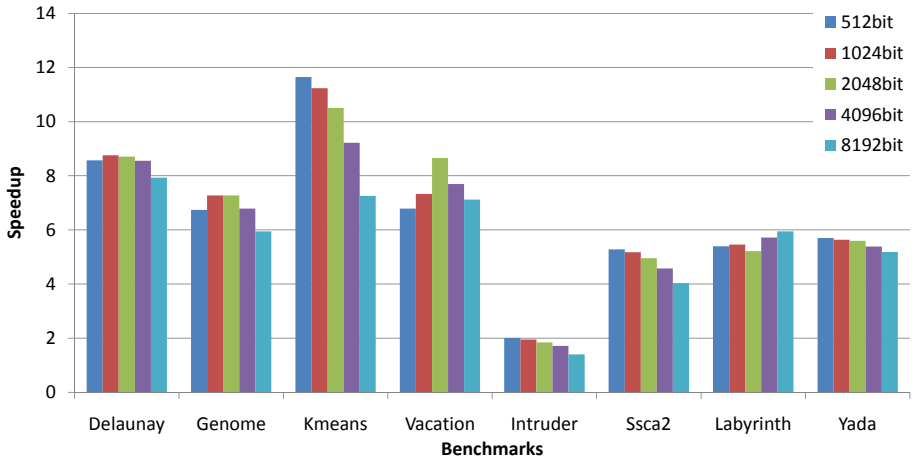


Figure 4.10: Sensitivity of BFGTS-HW to Bloom filter sizes ranging from 512bit-8192bit.

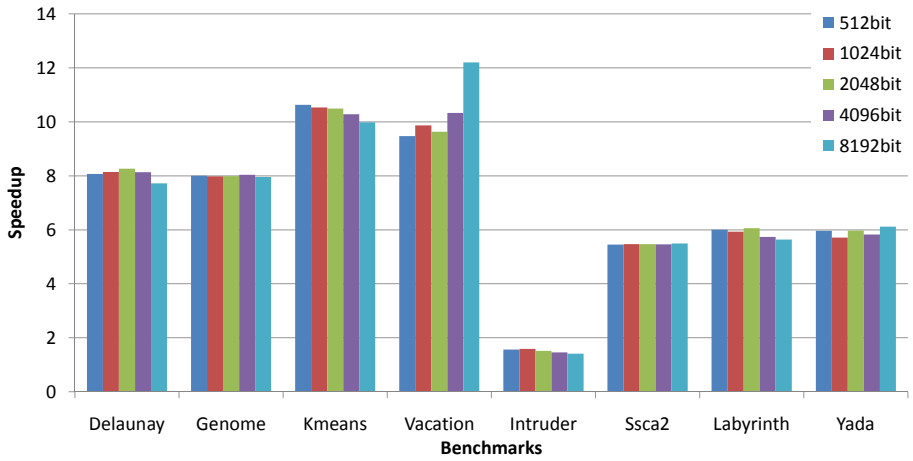


Figure 4.11: Sensitivity of BFGTS-SW/Backoff to Bloom filter sizes ranging from 512bit-8192bit.

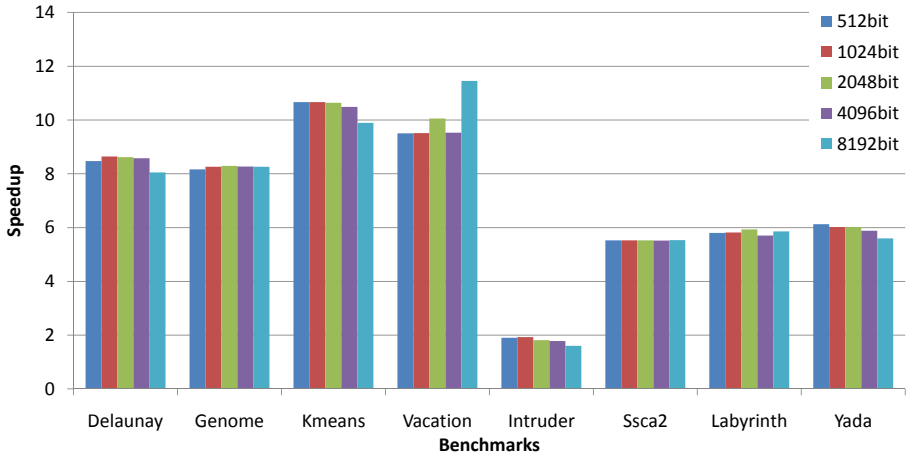


Figure 4.12: Sensitivity of BFGTS-HW/Backoff to Bloom filter sizes ranging from 512bit-8192bit.

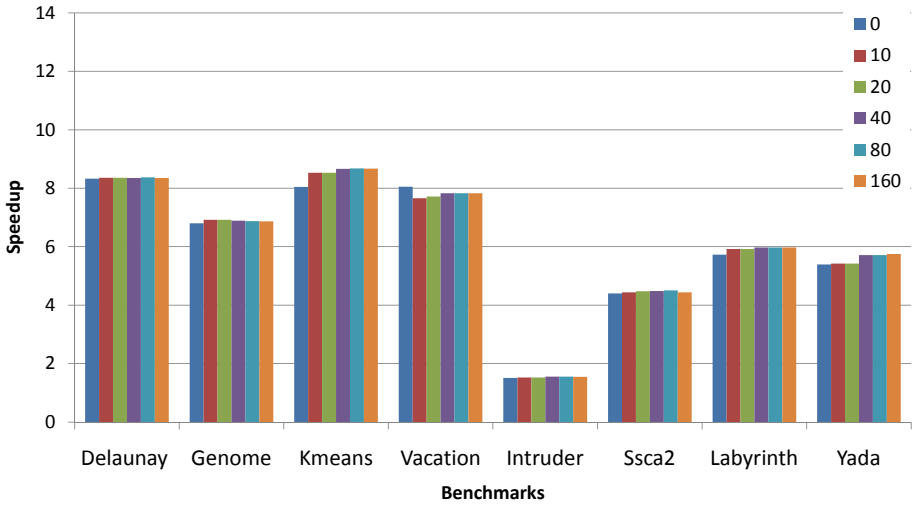


Figure 4.13: Sensitivity of BFGTS-SW to changing the frequency of updating small transaction similarity data every 0, 10, 20, 40, 80, 160 small transaction executions.

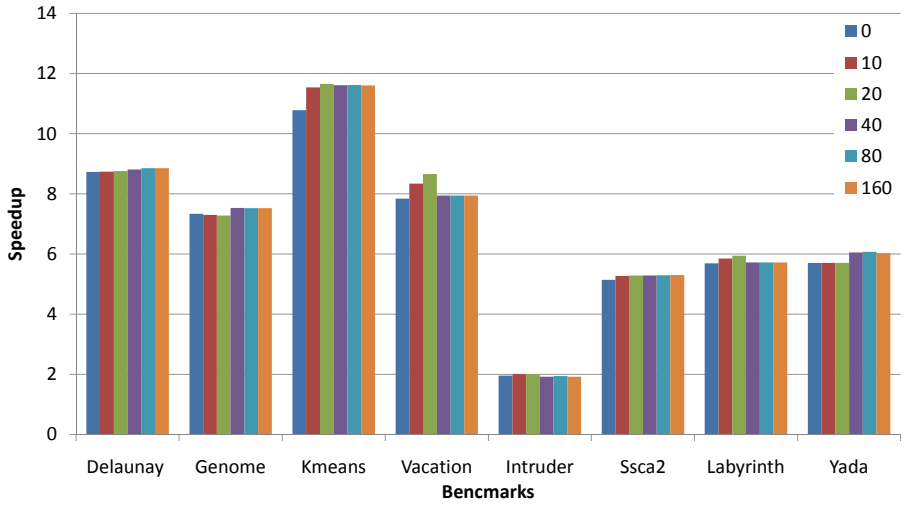


Figure 4.14: Sensitivity of BFGTS-HW to changing the frequency of updating small transaction similarity data every 0, 10, 20, 40, 80, and 160 small transaction executions.

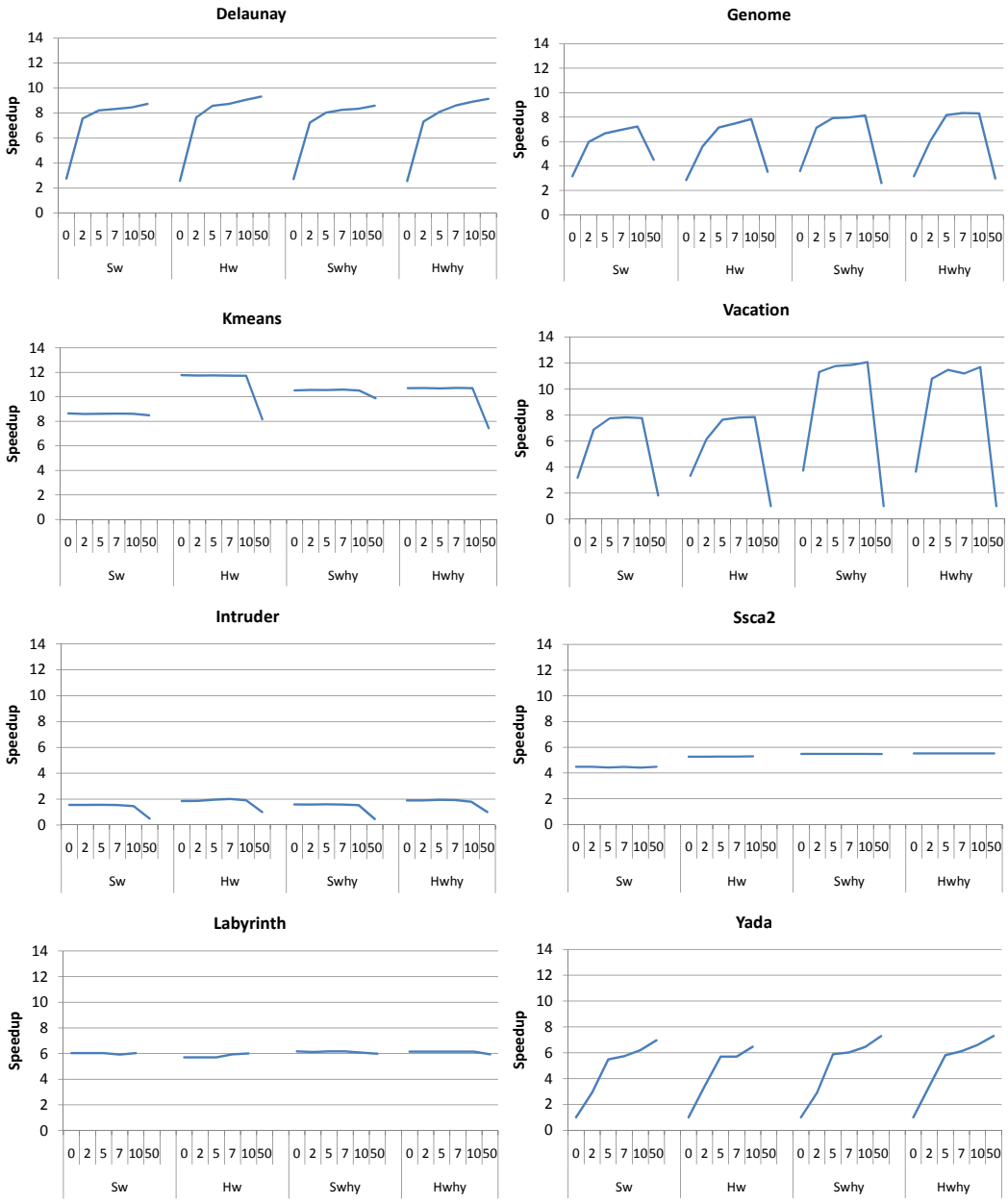


Figure 4.15: Sensitivity of BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff, and BFGTS-HW/Backoff to the time decay factor parameter set to 2, 5, 7, 10, and 50.

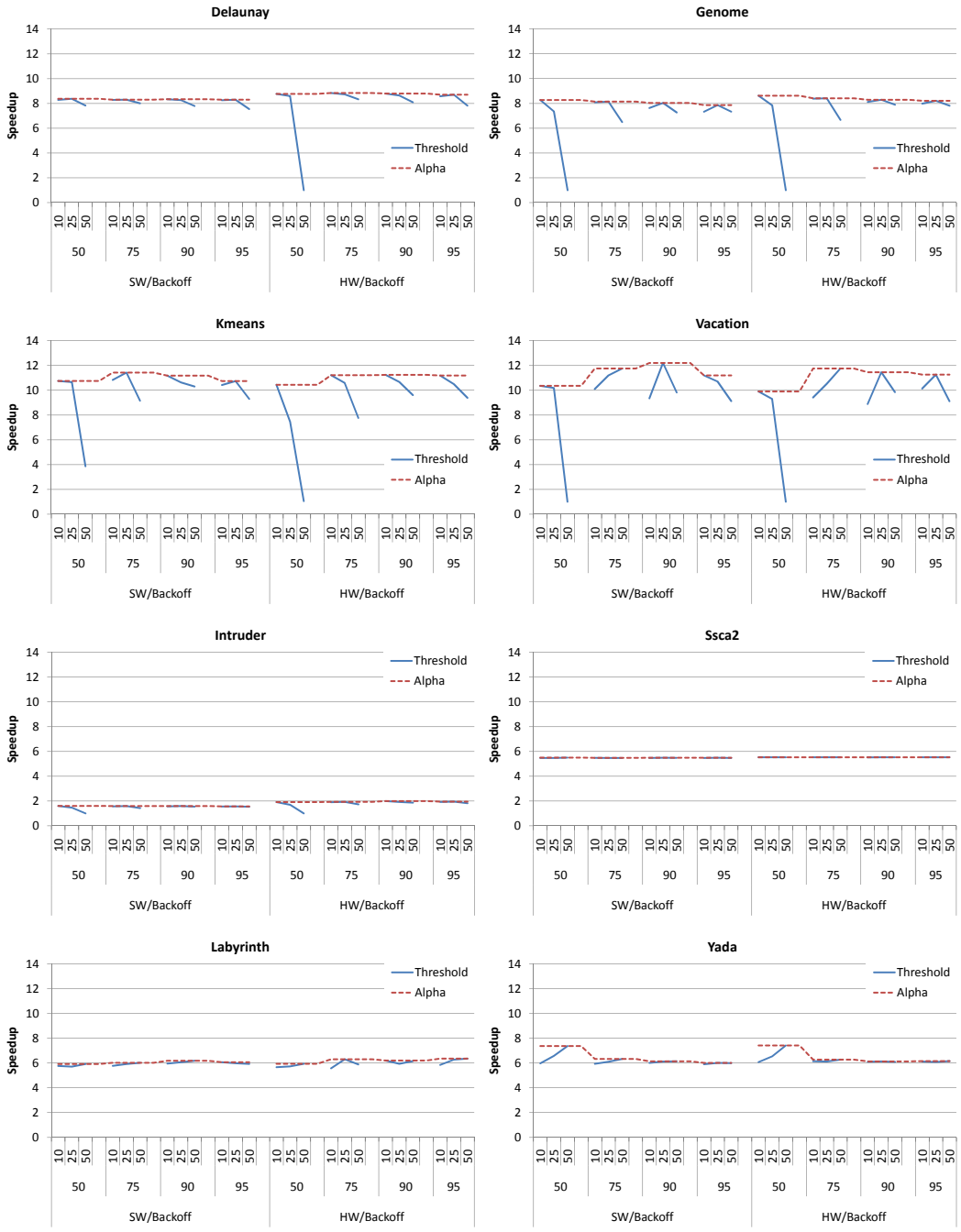


Figure 4.16: Sensitivity of BFGTS-SW/Backoff and BFGTS-HW/Backoff Hybrids to the switching threshold from backoff to BFGTS (solid line) and alpha value (dotted line).

CHAPTER 5

Voltage Boosting to Reduce Scalability Bottlenecks in Transactional Applications

This chapter describes the technique for determining which transactions are holding up other transactions (forming a scalability bottleneck) and the methods for boosting core frequencies to accelerate predicted bottlenecks. The circuit techniques and original architecture ideas originate from work done by Dreslinski et al. [52, 51]. This chapter is organized by first motivating the problem and the intuition behind the solution. Next the implementation details are described followed by an evaluation of the technique as it pertains to transactional applications.

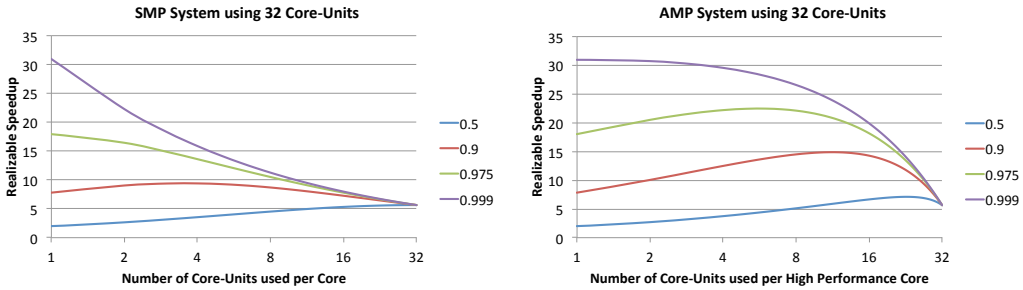
5.1 Asymmetric Multi-Processors

The power and complexity walls have effectively stopped the aggressive search for extraction of instruction level parallelism in single CPU cores. This stems from two problems, one is power. Power is a limiting factor when attempting to increase frequency of a processor, while designers can design the chip, the cooling requirements cannot be met. By increasing frequency, this also leads to more complex cpu cores to extract ILP by trying to find independent operations, speculate on branches that cannot be resolved immediately and tolerating increasing memory latencies as memory frequency has long been lagging the frequency of processor cores. This has met with diminishing returns in single threaded processor design. As described in Chapter 1, these problems have led to a resurgence in researching and development of parallel processor architectures. This research and development has culminated in chip-multiprocessors. But, parallel systems have their own major limitations, one serious limitation is described by Amdahl's equation reproduced below in Equation 5.1.

$$Speedup = \frac{1}{s + \frac{p}{n}} \tag{5.1}$$

As seen in Amdahl's equation, the limiting factor is the serial part s as it quickly dominates the equation when the number of processors n is large. Even though CMPs and parallel processing are considered **"the"** way to continue increasing performance, serial performance is still important. Studies by both Eyerman et al. [56] and Hill and Marty [68] show this to be true when analyzing the implications of Amdahl's equation to parallel processing and future architectures. They show that parallel processing gains quickly evaporate when the serial fraction is even a small percentage (1% or less) of the execution time. This serial fraction can be due to being just serial code, or contention on critical sections. This has led to the advocation of heterogeneous CMPs. Heterogeneous CMPs for general purpose computing combine many low performance cores for highly parallel sections of work, yet consume some chip area to implement at least one (or more) high performance cores to accelerate serial portions of code. Figure 5.1 shows a qualitative analysis of symmetric compared to heterogenous systems when the serial fraction is 0.1%, 2.5%, 10% and 50% of the execution time using r CPUs that use up chip area equivalent to 32 simple in-order CPUs (termed BCE's by Hill and Marty [68]) called "Core-Units" in this plot. As can be seen, the symmetric system has trouble gaining much performance when any serial code exists. One should note from the figure that as the amount of parallelizable code decreases, it is better to use fewer larger cores. The asymmetric system shows better scaling when devoting chip area to one or more large cores, but still using many little cores. The study by both Hill and Marty and Eyerman et al. are first order approximations that do not account for other effects, such as memory system organization, core interconnection and coherence policies. Regardless, these studies help motivate the necessary move to AMPs as a way to continue getting the benefit of increased transistor counts and transparently present programmers a way to get around scalability issues from serialization caused by critical section contention or other factors.

The main challenges with AMPs is the proper management of the cores. Having one or more cores that are more powerful than other cores pose problems with assigning threads to the right cores to optimize performance. Placing threads sub-optimally can lead to worse than expected performance. This chapter provides a potential solution to determining the proper threads to schedule to the more powerful cores in the context of transactional applications and using the prediction methods from the previous two chapters to determine that proper schedule. As seen in the previous chapters the benchmarks in the STAMP suite have numerous points of serialization that lead to degraded performance. This degraded



(a) Symmetric System

(b) Asymmetric System

Figure 5.1: Speedup's attainable on an area equivalent of 32 simple in-order cores (called Core-Units) SMP and AMP systems core system with 99.9%, 97.5%, 90% and 50% parallelizable code(figures reproduced from equations in paper by Hill and Marty [68]

performance can be partially gained back by using an AMP system. In this chapter I propose using the prediction techniques developed in the previous chapters along with a novel AMP system proposed by Dreslinski et al. that uses voltage boosting to increase frequency to effectively move the "Large Powerful" core to the code that needs accelerating. This is substantially different from past works such the ACS work by Suleman et al [103] that assumes a fixed system with one large core where code and data have to be migrated to be accelerated.

5.2 Implementation

5.2.1 Voltage Boosting Architecture

There are three architecture designs that will be evaluated in this chapter to investigate the concept of using voltage boosting of the bottle-necking core/thread to create a dynamically configuring AMP system that can accelerate transactional applications with high contention.

The main concept for these architectures is to design the architecture for low voltage and frequency operation with the capability to boost to higher voltage and frequencies when necessary. The reasoning behind this instead of the traditional designs that design for performance first then include the capability to scale down the voltage is that future parallel codes will be able to operate at lower frequencies because they can scale out to numerous cores, and only need to occasionally use high frequency operation. This allows for a more efficient design in terms of power and energy. Using voltage boosting for making

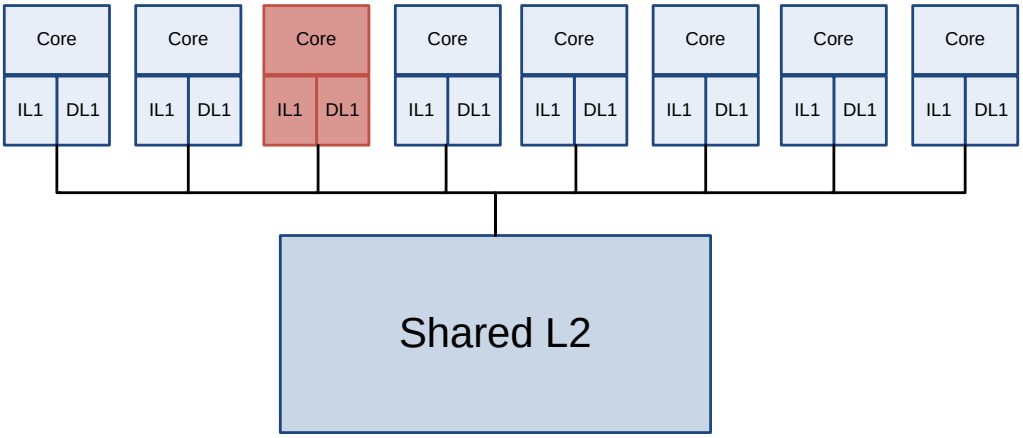


Figure 5.2: Boosting one core (shown in red) while the remaining cores operate at less than nominal voltage (light blue) to keep within TDP.

an AMP also has an advantage over previously proposed AMPs is the core being boosted can move. This removes the need to move data around the chip to the more powerful core when sequential execution is needed.

These boosting enabled architectures are assumed to be using two voltage rails to supply the CMP with a low voltage and a high voltage. Cores can then switch between the two rails to operate at the low frequency or the high frequency. The dual rail architecture can theoretically switch between frequencies in 10's of cycles instead of the 10's of thousands of cycles it takes with traditional DVFS techniques. More information on the circuit techniques used to create logic and SRAM that operates at low voltage and frequency and fast switching between voltage rails can be found in work by Dreslinski et al. [52, 51, 114]. This section will describe the architectures and the trade-offs inherent in each.

Figure 5.2 shows a voltage boosting architecture that allows at most one core and its L1 caches to boost to the higher frequency by switching the voltage rail being used. To maintain the same power dissipation as a homogeneous CMP, the low frequency configuration must run the rest of the cores at a slightly slower speed to allow for boosting one core and keeping the other cores active. This allows potentially for the maximum amount of parallelism to be maintained while boosting at most one core. The downside to this configuration is the requirement to operate the cores at a lower frequency to accommodate the boosting core. This can potentially lead to less performance in the absence of any scalability bottlenecks compared to a homogeneous design.

Figure 5.3 shows an alternate architecture called the "Cluster" boosting architecture. In this architecture, all the cores run at the same nominal frequency as the homogeneous architecture. To allow boosting, three out of every four cores must be shut down to allow

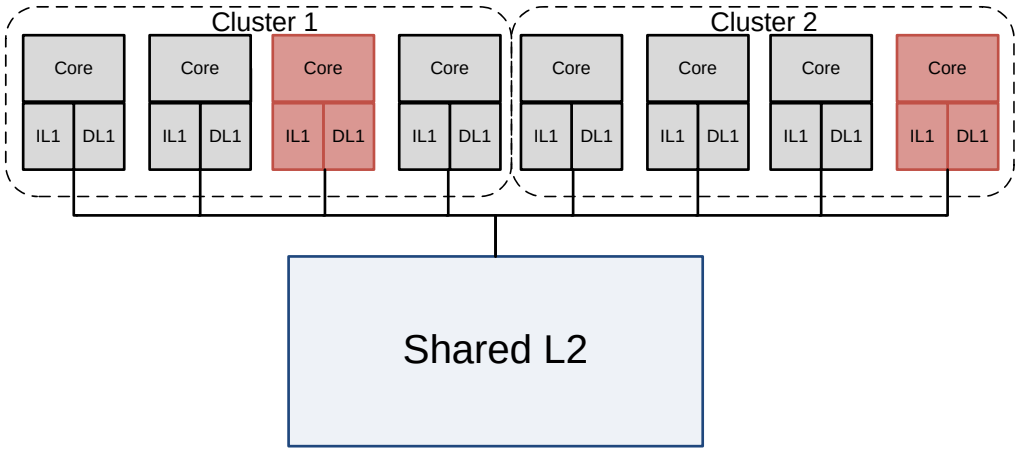


Figure 5.3: Cluster Boosting architecture showing two clusters. One cluster runs at nominal voltage (blue), while the other allows one boosting core (red) with the others shut off (gray), caches remain active to serve coherence requests.

one core to boost to a higher frequency along with its L1 caches. The L1 caches of the inactive cores remain powered to allow for cache coherence requests to be served. This allows for this system to maintain performance when there are no scalability bottlenecks, and allow more than one boosted core during execution. The drawback to this architecture design is it may limit maximum attainable performance by having to disable cores to allow boosting as well as not making use of the idle cache space left active when cores are shut off.

The final design shown in Figure 5.4 and is called the "NTC" architecture. It takes full advantage of the properties of near-threshold circuit design. When operating at lower voltages, logic and SRAM have different optimal voltages. SRAM can operate faster than logic as seen by Zhai et al. [114]. This allows one to design the system in Figure 5.4 where caches can be shared among clusters of cores. In this case the cache is running faster than the cores and can therefore be effectively shared when running at low voltages. As with the "Cluster" boosting architecture, only one core in a cluster can be boosted when the other cores in the cluster are turned off. Unlike the "Cluster" architecture, the NTC architecture gets full use of the cache space when in boosted mode. A potential problem is cache pollution from the boosted core when it gets full access to the shared cache.

5.2.2 Identifying Problem Critical Sections

The key challenge in designing AMPs is categorizing and scheduling threads to the correct cores to get the optimal runtime. This is a hard challenge and there has been lots

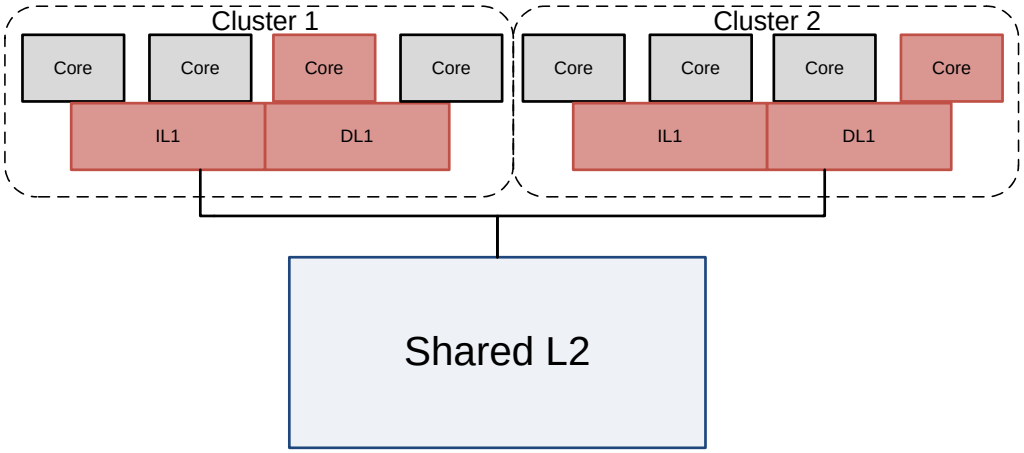


Figure 5.4: NTC Boosting architecture showing two clusters of cores. Unboosted cores (blue) share a large cache operating at high frequency (red). Boosted cores (red) get access to the entire shared cache but the remaining cores in a cluster must be shut off (gray).

of work in investigating thread scheduling for AMPs. Work in this area includes work by Lakshminarayana et al. [76], Balakrishnan et al. [23] and Li et al. [80] that looks at thread scheduling in AMPs from the OS and program perspective. Another main challenges for AMPs is reducing the overhead of migrating threads as their execution can be phasic, sometimes requiring a slow core, sometimes requiring the fast core. Using Dreslinski's voltage boosting technique, this is no longer a concern as the core can move to the thread and its data. Still, the challenge remains with determining which core and its associated thread needs to be boosted.

Transactional applications are an interesting group of applications to study in the AMP context. As seen in the previous chapters, these applications can be susceptible to large amounts of contention that slow down execution when programmed using large transactions in a manner that less expert programmers may use. Chapter's 3 and 4 presented scheduling methods to reduce scalability bottlenecks to tolerate the latency imposed by them. In this chapter I present using the techniques from those chapters to derive an algorithm to determine the proper core to boost for three architecture design points that leverage boosting to create an AMP. This helps to further reduce scalability bottlenecks.

5.2.2.1 Boosting One Core Algorithm

The algorithm for the "Boost One" architecture, is shown in Example 5.1. On every *Transaction Begin*, *Transaction Abort* and *Transaction Commit* credits are modified. For

Transaction Abort and *Transaction Commit*, credits for the local CPU are reset to zero to signify this particular core does not need to be boosted as it is not running within a critical section. On *Transaction Begin*, the BFGTS algorithm is used to predict whether if the transaction needs to serialize. If a serialization is predicted to be required, credits are added to the transaction/core that is predicted to cause an abort. When the credits are modified, the system evaluates whether a core needs to be boosted, or an already boosting core needs to relinquish its boost and a different core should boost. This is determined by looking for the oldest running transaction with the most credits assigned to it. Looking for the oldest transaction allows for fair boosting. To determine which core should boost is a centralized operation, and although not explored in this thesis could be implemented as a central hardware structure. Unfortunately this will not scale to larger core counts, and may be a limiting factor to future CMPs that are constrained to boosting only one core.

Example 5.1 TM Boosting Pseudo-code used to evaluate when to boost a core for the "Boost One" architecture

```
1  void evaluateCoreToBoost()
2  {
3    boost_cpu = -1
4    max_credit = 0
5    time_stamp = 0xffffffffffffffffULL;
6
7    for (i=0; i < numCpus; i++) {
8      if (cpuArray[i].credits > max_credit
9          && cpuArray[i].time_stamp < time_stamp) {
10       boost_cpu = i;
11       max_credit = cpuArray[i].credits;
12       time_stamp = cpuArray[i].time_stamp;
13     }
14   }
15
16   if (max_credit > 0
17       && boost_cpu > -1) {
18     boostCPU(boost_cpu);
19   }
20 }
```

5.2.2.2 Cluster and NTC Boost Algorithm

Boosting decisions for the "Cluster" and "NTC" architectures requires a more complicated algorithm to decide when to boost. As with the "Boost One" algorithm, cores are given credits as other cores predict they have to serialize their transactions. As with the

"Boost One" architecture, determining when to boost is done on *Transaction Begin*, *Transaction Abort* and *Transaction Commit*. It is also done when a transaction is signaled it no longer needs to serialize behind another transaction.

Example 5.2 TM Boosting Pseudo-code used to evaluate when to boost a core for the "Cluster Boost" and "NTC" architectures when a core is beginning a transaction or stalling.

```
1  void evaluateCoreToBoost(int cpu, int num_stalled)
2  {
3    max_credits = 0;
4    boost_candidate = -1;
5
6    // Check for a boosting candidate via credits
7    for (i = 0; i < cluster_size; i++) {
8      if ((clusterArray[i].credits >= max_credits) &&
9          !clusterArray[i].stalling) {
10       max_credits = clusterArray[i].credits;
11       boost_candidate = i;
12     }
13   }
14
15   // relinquish our opportunistic boost if we are stalling
16   if (clusterArray[cpu].stalling) {
17     if (clusterArray[cpu].boosting) {
18       clusterArray[cpu].boosting = false;
19       unboost_clock(cpu);
20     }
21     if (num_stalled == (cluster_size-1)) {
22       clusterArray[boost_candidate].boosting = true;
23       boost_clock(boost_candidate);
24     }
25   } else if (num_stalled == (cluster_size-1) &&
26       !clusterArray[cpu].boosting) {
27     // Opportunistically boost
28     clusterArray[cpu].boosting = true;
29     boost_clock(cpu);
30   } else if (num_stalled == (cluster_size-2)
31              && clusterArray[boost_candidate].credits
32                 >= boost_threshold) {
33     // predicted to not stall, but relinquish voluntarily.
34     clusterArray[boost_candidate].boosting = true;
35     boost_clock(boost_candidate);
36     clusterArray[cpu].stalling = true;
37   }
38 }
```

Example 5.2 shows the pseudo-code algorithm used to determine when to boost a core

when the local core is either predicted to have no conflict and execute or is going to stall. As it can be seen, this pseudo-code is more complicated than the pseudo-code used in the previous section. This is from the added complication of only being able to boost when all the other cores in the cluster of CPUs are turned off. There are three conditions that this algorithm boosts a core for: 1) A core is stalling to serialize behind a predicted conflict, release its boost (if applicable) and try to boost another core if all but one remaining core are active, 2) If the core is predicted to have no conflicts, then see if it can opportunistically boost if the other cores in the cluster are inactive 3) The core is predicted to have no conflicts but another core has a large number of credits greater than some threshold, then it will stall sacrificially to allow the other core to boost. As can be seen from the code and the conditions listed, there are potentially less opportunities to boost in this configuration as having the most credits is no longer enough to get boosted. A core must either have many credits to force a core to sacrifice itself as well as have other cores inactive in its cluster, or all the cores are already inactive to allow for an opportunistic boost. Even though having to shut down cores to allow boosting complicates matters, one benefit that could be taken advantage of is that this algorithm can be distributed among clusters as cores only boost by looking at information local to the cluster. This makes this architecture more scalable to more cores.

Example 5.3 shows what happens when a core wakes up from stalling behind a predicted conflict. It first checks to see if it can opportunistically boost if the other cores happen to be inactive in its cluster. If boosting is not possible, it takes away the boost of another core if applicable as more than one core are becoming active in a cluster. Credits are not checked here because the system is biased towards running in parallel instead of keeping a boosted core running fast.

One thing to note from these two pieces of pseudo-code for the "Cluster" based boosting algorithm is that cores are only allowed to determine when to start and end boosting along the boundaries of a transaction. This guarantees that a processor is not in code protected by a lock and then accidentally shut down if another core wants to boost potentially causing a dead-lock. This may be pessimistic, but this chapter is showing just one implementation of boosting. Optimizations to the boosting algorithms are left for future work.

5.3 Evaluation

5.3.1 Simulation Environment and Benchmarks

To test voltage-boosting for transactional applications I again use the STAMP suite to evaluate. The benchmark parameters are shown in Table 5.1. The main difference to

Example 5.3 TM Boosting Pseudo-code used to evaluate when to boost a core for the "Cluster Boost" and "NTC" when a core is waking up from being shut down.

```
1  void evaluateCoreToUnBoost(int cpu, int num_stalled)
2  {
3    boost_core = -1;
4
5    for (i = 0; i < cluster_size; i++) {
6      if (clusterArray[i].boosting) {
7        boost_core = i;
8      }
9    }
10
11   // opportunistically boost
12   if (num_stalled == 3 &&
13       !clusterArray[cpu].boosting) {
14     clusterArray[cpu].boosting = true;
15     boost_clock(cpu);
16   } else if (num_stalled < 3 && boost_core != -1) {
17     // Else, unboost as more cores are coming online
18     clusterArray[boost_core].boosting = false;
19     unboost_clock(boost_core);
20   }
21   return;
22 }
```

be noted is the benchmarks are configured to use the same number of cores as threads. This was done for two reasons: 1) To investigate the maximum gain voltage boosting can attain when cores have to idle when a conflict is predicted and 2) the "Cluster" and "NTC" architectures need to turn off cores, and this can not easily be done when there are more threads than cores. I will show results for the "Boost One" architecture using more threads than cores to see the benefits of boosting in the overcommitted configuration where BFGTS can switch in another thread and also use boosting.

For these tests I again use the M5 Full System simulator with LogTM support. The contention manager used in BFGTS-NoOverhead to measure the impact of voltage-boosting only on performance. The frequency configurations are taken from Dreslinski et al. [52] and are derived from SPICE simulations of ARM-Cortex M3 cores running at NTC voltages. The relative frequency ratios are 4x, 2.7x, and 1.6x for the "Boost One" architecture. For the "Cluster" and "NTC" boost architectures, they are tested with the base frequency of 320MHz and boost frequencies of 750MHz (2.7x) and 1280MHz (4x). The "NTC" architecture assumes you can build a 256kB L1 cache that can be accessed in one cycle. This may be unrealistic, but it was decided to be unfair to penalize the "NTC" system by giving

Benchmark	Input Parameters
Delaunay [72]	-i large.2 -m30 -t16
Genome	-g4096 -s32 -n524288 -t16
Kmeans	-m20 -n20 -t0.05 -i random50000_12 -p16
Vacation	-n8 -q10 -u80 -r65536 -t131072 -c16
Intruder	-a10 -l32 -n8192 -s1 -t16
Ssca2	-s15 -i1.0 -u1.0 -l3 -p3 -t16
Labyrinth	-i random-x96-y96-z3-n128.txt -t16
Yada	-i large.2 -m30 -t16

Table 5.1: STAMP Benchmark input parameters.

Feature	Description
Processors	16 one IPC Alpha cores @ 250MHz, 275MHz, 310MHz and 320MHz
Boost Frequencies	1000MHz, 750MHz, 500MHz, 1280MHz
L1 Caches	64kB or 256kB, 1 cycle latency, 2-way associative, 64-byte line size
L2 Cache	32MB, 6 cycle latency to unboosted core frequency, 16-way associative, 64-byte line size
Interconnect	Shared bus @ 250MHz, 275MHz, 310MHz, and 320MHz
Main Memory	2048MB, 50ns latency
Linux Kernel	Modified v2.6.18
Contention Managers	BFGTS-NoOverhead with Boosting Extensions
Signature Size	Perfect signatures used for conflict detection and BFGTS-NoOverhead.

Table 5.2: M5 Simulation Parameters.

it less cache space. The L1s can run at either the unboosted, or boosted frequencies with a 1 cycle latency. The L2 speed is fixed at a 6 cycle latency to the unboosted frequency. The baseline homogenous configuration runs at a speed of 320MHz, with the same cache access latency ratios, but does not use voltage boosting. One thing to note here is that the memory latencies are similar to the previous chapters, but the core frequencies are lower resulting in a memory system that looks closer to the cores.

5.3.2 Performance Analysis

This section presents a performance analysis of the three boosting architectures and organized as follows. I first present results for a non-overcommitted system using the "Boost One" architecture. Then I will present results for the "Cluster" and "NTC" boosting architectures.

Figure 5.5 shows the overall speedup of each frequency of the boost-one architecture for 16 processors over 1 core running at 320MHz. Figure 5.5 shows the performance attained

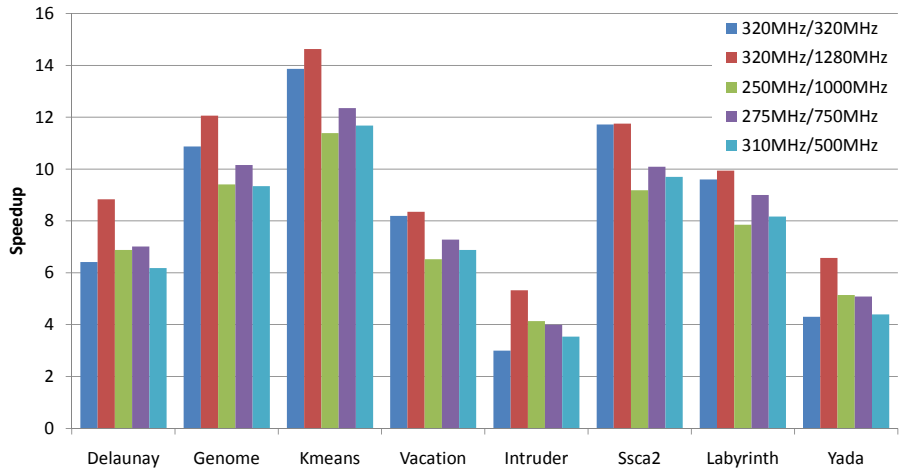


Figure 5.5: Speedup for a non-overcommitted 16 processor system for boosting one core architecture.

for the homogeneous system (320MHz/320MHz), an ideal 4x boost (320MHz/1280MHz), 4x boost (250MHz/1000MHz), 2.7x boost (275MHz/750MHz) and a 1.6x boost configuration (310MHz/500MHz). Figure 5.6 shows all the tested configurations normalized to the homogeneous CMP configuration. As can be seen from Figures 5.5 and 5.6 the performance of the "Boost One" architecture improves performance for 3 out of 8 benchmarks when looking at the iso-power configurations, and decreases the performance for the remaining benchmarks. On average, this leads to overall performance being slightly worse than a homogeneous system. The ideal 4x boost case shows equivalent or better performance for all the benchmarks and can achieve a $>$20% performance improvement as it can run parallel code as fast as the homogeneous system.

The "Boost One" configuration can be better understood by also looking at Table 5.3. For the benchmarks where "Boost One" loses performance (*Genome*, *Kmeans*, *Vacation*, *Ssca2* and *Labyrinth*), it can be seen that boosting is used less than 20% of the time for these benchmarks. This shows that there are not many scalability bottlenecks for boosting to recover performance from. This fact immediately puts the "Boost One" architecture at a disadvantage as the cores are running at a reduced frequency and therefore will not get as good performance as the homogeneous system. For the three benchmarks where boosting is advantageous (*Delaunay*, *Intruder*, and *Yada*), boosting can attain up to a 40% speedup for the case of the *Intruder* benchmark. Looking at Table 5.3 clearly shows why. These benchmarks see a greater than 50% of the time with one core in the boosted state. This is not unexpected as these benchmarks have a large amount of contention as seen from the previous chapters. Because these benchmarks have such a large number of bottlenecks that

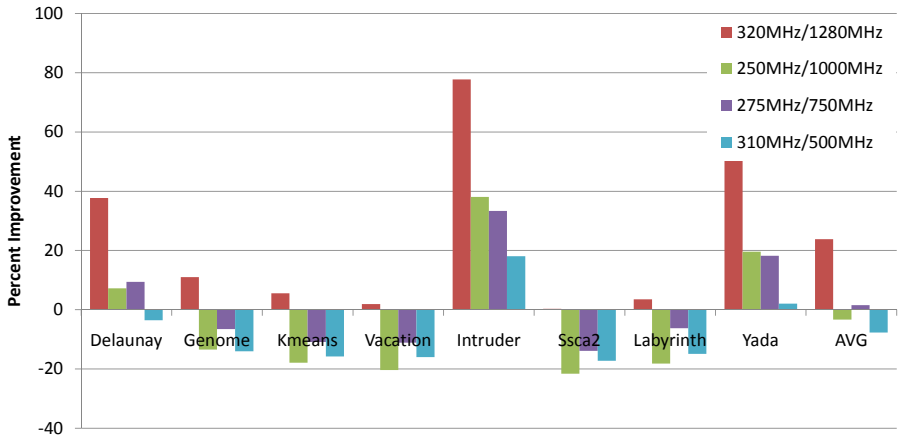


Figure 5.6: Percent difference in speedup for a non-overcommitted 16 processor system for boosting one core architecture compared to a non-boosted architecture.

benefit from boosting, they also prefer the 4x boosting configuration with the 2.7x boosting configuration loosing a small amount of performance. The 1.6x boosting configuration performs the worst as it is not able to accelerate the bottlenecks enough to make up for the reduced parallel performance. Overall the "Boost One" architecture requires too severe a trade-off to efficiently accelerate scalability bottlenecks where parallel performance is reduced to allow one core to boost up to 100% of the time. In practice no core stays boosted for 100% of the time.

Figures 5.7 and 5.8 show speedup and percent difference over the homogeneous system for the "Cluster" and "NTC" architectures. I experiment with two frequency configurations, an iso-power configuration for the "Cluster" and "NTC" architectures (320MHz/750MHz) and an ideal 4x boosting configuration where the "NTC" architecture sees its shared L1 cache running at a speed sufficient to feed all cores in one clock cycle when they are not boosted.

As can be seen from the "Cluster" configuration gets slightly less performance on the three benchmarks (*Delaunay*, *Intruder* and *Yada*) than the "Boost One" architecture, but is able to get better or equivalent performance for the rest of the benchmarks where boosting is less effective. This is because for mostly parallel benchmarks the "Cluster" architecture can run the unboosted cores at the same frequency as the homogeneous system. For the benchmarks showcasing scalability bottlenecks, the "Cluster" architecture cannot quite extract the same amount of performance as the "Boost One" architecture due to the less opportunity to boost with the boosting algorithms presented that favor running parallel over boosting. This can be seen Table 5.4. The amount of time spent boosting for the "Cluster"

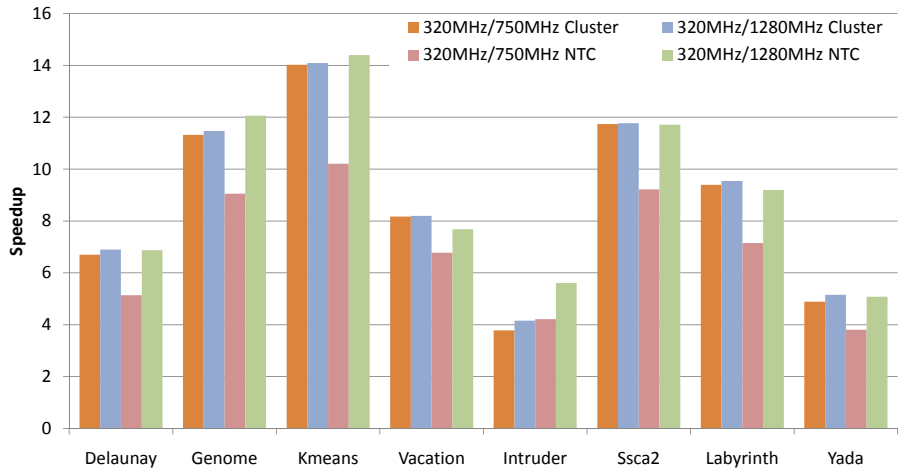


Figure 5.7: Speedup for a non-overcommitted 16 processor system for cluster and NTC boosting architectures architecture.

architecture is substantially less than the "Boost One" architecture. Overall the "Cluster" architecture strikes a better balance than the "Boost One" architecture. It can run the cores at higher frequency when not voltage boosting to regain parallel performance yet still boost effectively enough to accelerate bottlenecks. When the "Cluster" architecture is given an ideal 4x boost, it can attain even higher performance as seen in the figures.

The "NTC" architecture is interesting to look at for a comparison point. When run at an iso-power point of 320MHz/750MHz, only one benchmark gets improved performance, the *Intruder* benchmark. The *Intruder* benchmark has many bottlenecks, as seen from Table 5.3. This allows the "NTC" architecture to get better performance than the homogeneous architecture. Interestingly enough, it gets even more performance than the ideal 4x boost "Cluster" architecture. This is due to the large L1 cache available to the boosted core running at boosted speed compared to the small boosted L1 available to the "Cluster" architecture. On the other hand, because the L1 cache is running slower than 4x the frequency of the cores, the "NTC" architecture does not run as fast due to being bandwidth limited to the L1 cache and loses significant performance for all the other benchmarks. For the "NTC" architecture to achieve good performance it must have an L1 cache with sufficient bandwidth. When running with an ideal 4x boost, the "NTC" architecture gets the best performance overall as seen in Figures 5.7 and 5.8. The *Genome* and *Intruder* both see performance improvements due to the shared L1 cache architecture. The *Yada* and *Vacation* benchmarks see small performance losses over the "Cluster" architecture most likely due to cache pollution affects.

As seen from this section boosting architectures can be profitably leveraged to acceler-

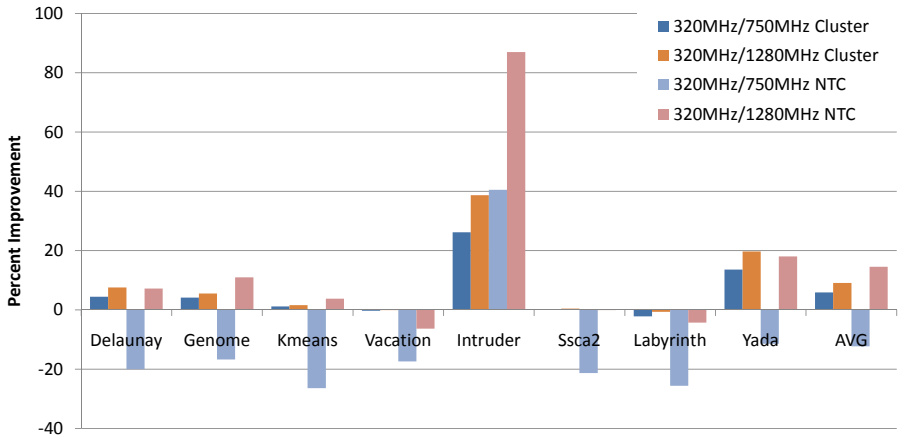


Figure 5.8: Percent difference in speedup for a non-overcommitted 16 processor system for cluster and NTC boosting architectures compared to a non-boosted architecture.

ate transactional applications when bottlenecks can be accurately identified. The "Cluster" and "NTC" architectures are the clearly better design points as they do not have performance degradations for largely parallel codes, yet can still boost effectively. Future algorithm enhancements may be able to extract more performance from the "Cluster" and "NTC" architectures.

5.3.3 Sensitivity Studies

To fully evaluate voltage-boosting for transactional applications, two sensitivity tests were conducted. The first was varying how long it takes to switch voltage rails from low to high voltages for the different architectures. The second takes a look at the effects of boosting on an overcommitted system that can also take advantage of switching in different threads to find additional parallel work.

5.3.3.1 Boosting Latency

The results presented in the previous section were taken assuming the latency to switch from low to high voltage was zero. A zero latency switch is the most optimal switch latency for all types of dynamic voltage and frequency scaling systems. A zero cycle latency is not feasible. Therefore I look at how sensitive the presented voltage boosting architecture is for the tested benchmarks.

Figure 5.9 shows the normalized average performance over all the benchmarks to a

	Delaunay	Genome	Kmeans	Vacation
250MHz/1000MHz	71.3%	14.8%	3.8%	15.2%
275MHz/750MHz	78.1%	16.9%	5.1%	15.8%
310MHz/500MHz	84.1%	19.5%	5.2%	20.5%
320MHz/1280MHz	71.3%	14.7%	3.7%	15.2%
320MHz/750MHz Cluster	31.6%	18.5%	3.9%	7.8%
320MHz/1280MHz Cluster	24.8%	15.7%	3.0%	5.9%
320MHz/750MHz NTC	29.1%	14.2%	0.1%	8.3%
320MHz/1280MHz NTC	26.1%	13.8%	0.0%	7.7%
	Intruder	Ssca2	Labyrinth	Yada
250MHz/1000MHz	57.2%	0.0%	3.3%	65.2%
275MHz/750MHz	63.1%	0.0%	2.7%	71.8%
310MHz/500MHz	69.7%	0.0%	6.4%	78.4%
320MHz/1280MHz	56.8%	0.0%	3.0%	65.1%
320MHz/750MHz Cluster	78.6%	0.0%	2.3%	46.9%
320MHz/1280MHz Cluster	63.4%	0.0%	0.6%	38.4%
320MHz/750MHz NTC	76.4%	0.0%	4.4%	44.0%
320MHz/1280MHz NTC	68.4%	0.0%	2.9%	40.5%

Table 5.3: Percentage of execution time that a core was in a boosted state for a non-overcommitted system with 0 cycle boosting latency

zero cycle latency for each architecture as boosting latency is varied. The boost latencies tested were 5,10,25,50,100 and 1000 cycle boost latencies. Figure 5.9 shows that the boosting architectures can tolerate boosting latencies of around 100 cycles of latency, seeing a 10% performance degradation. At 1000 cycles, over a 30% performance degradation is observed. This makes sense as in the previous chapters the average length of a transaction was between a few 10's and 1000's of cycles in length. If the boosting latency was greater than 1000 cycles, a transaction that needed to be boosted could complete before the boosting completed. Fortunately, as seen in Dreslinski's thesis [52], 10's of cycles of boosting latency is feasible with the proper circuit design.

5.3.3.2 Overcommitted System

The final sensitivity test looks at how effective voltage boosting is when a transactional system is using the full potential of its scheduling contention manager by allowing threads to be switched in and out to find independent work and avoid stalling. Testing with equal numbers of threads to cores allows the effect of bottlenecks to be exposed fully, but using equal number of threads to cores is in fact not a common occurrence [29].

Figure 5.10 shows the speedup for a 16 processor overcommitted system using 64 threads for the STAMP benchmarks using BFGTS-NoOverhead and the "Boost One" sys-

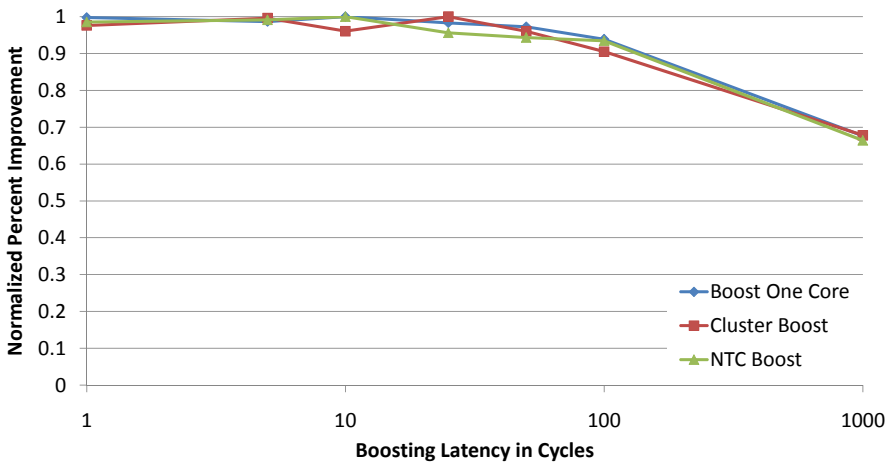


Figure 5.9: Boost latency sensitivity for a non-overcommitted 16 processor system for the three boosting architectures.

tem. Table 5.4 shows the percentage of time a core spends boosting. The "Cluster" and "NTC" architectures are not tested because currently they need to shut down cores to be able to boost and no algorithm has been designed yet to allow a boosting scheme for these architectures that can guarantee deadlock freedom. As can be seen in the figure, an overcommitted system sees less benefit from a boosting system. This is because the effects of scalability bottlenecks are less severe when a system can run other threads in place of a blocked thread. This in effect hides the latency of the bottleneck much like an Out-of-Order processor hides long latency operations by executing independent work. All but two of the benchmarks run best on the homogenous system. The *Intruder* and *Yada* benchmarks still see a benefit as the bottlenecks still exist because the scheduling of other threads is not completely effective. In these cases the 4x boosting system gets the best performance because it can clear a bottleneck the fastest. If an ideal 4x boosting system is used then again all benchmarks see a benefit from boosting. If an algorithm was designed for the "Cluster" and "NTC" architectures to tolerate thread switching, then they may see better results than the "Boost One" architecture as was seen in the previous sections.

5.4 Conclusions

This thesis offers a preliminary evaluation of using voltage boosting to accelerate transactional applications that have scalability bottlenecks. As seen boosting can offer better performance than a homogeneous system using the BFGTS scheduler, especially if the system is not overcommitted. The best architecture was the "NTC" architecture when the

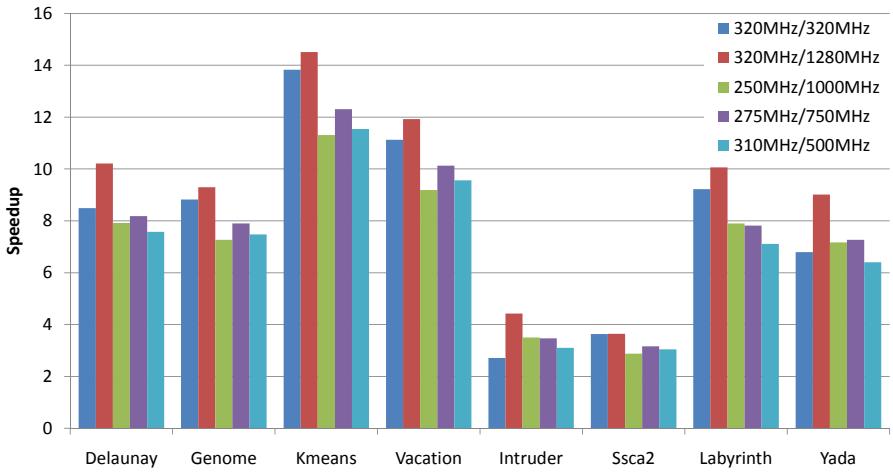


Figure 5.10: Sensitivity of the boosting one core architecture for an overcommitted 16 processor system.

	Delaunay	Genome	Kmeans	Vacation
250MHz/1000MHz	86.4%	12.5%	2.7%	12.1%
275MHz/750MHz	90.2%	14.4%	3.8%	14.2%
310MHz/500MHz	93.3%	16.4%	4.1%	17.7%
320MHz/1280MHz	86.6%	12.5%	2.7%	12.5%
	Intruder	Ssca2	Labyrinth	Yada
250MHz/1000MHz	48.7%	0.0%	12.9%	73.0%
275MHz/750MHz	55.2%	0.0%	13.1%	80.2%
310MHz/500MHz	62.1%	0.0%	11.9%	87.0%
320MHz/1280MHz	49.2%	0.0%	14.8%	74.3%

Table 5.4: Percentage of execution time that a core was in a boosted state for a overcommitted system using a 0 cycle boost latency

shared caches could support the bandwidth demands of the connected cores. Otherwise the "Cluster" architecture is the best balance point as it can run parallel benchmarks full speed but still accelerate severe bottlenecks, whereas the "Boost One" architecture can boost more but it sacrifices too much parallel performance to remain iso-power. Overall, at an iso-power operation point boosting can attain a 15% performance improvement over a homogenous architecture on average. Ideally boosting can get greater than a 50% improvement on average.

There is a more work that needs to be done in this area. The boosting algorithms need more development, especially for the "Cluster" and "NTC" architectures to find the correct balance between boosting and parallel operation. These architectures also need to be

re-designed to accommodate an overcommitted system, as it achieves better performance because it does not waste as many processor resources stalling. Currently these architectures cannot support this type of operation without potentially causing deadlock.

CHAPTER 6

Multi-Threaded Fetch Throttling in Transactional Applications

This chapter describes a technique for guiding fetch bandwidth allocation in MT processors that have support for HTM. Scalability bottlenecks that exist in MT processors also need to be accelerated, but it must be tackled differently than in the previous chapter. This chapter is organized by first motivating the problem, discussing the implementation details and evaluating the technique to guide fetch throttling to accelerate bottlenecks and increase performance. This work is an evaluation of a proposal published in a patent application by Blake et al. [31].

6.1 Motivation

On CMPs the effect of memory latency can decrease the efficiency of a single core because multiple cores are accessing the memory sub-system concurrently. This is because the memory sub-system has not scaled at the same rate as number of cores placed on die in terms of access latency and available bandwidth per core. This effectively decreases the available memory resources to each core, increasing memory access latency, leading to the core stalling more often than it would otherwise. The architectural feature of multi-threading is now used to increase efficiency of CMPs. MT allows greater throughput as a core's resources are better utilized by multiple thread's instructions and helps to hide the effects of memory latency. MT accomplishes this by allowing more threads to execute concurrently than there are physical cores by sharing resources on a single core between multiple threads. Several commercial architectures now use multi-threading to increase throughput as there are more pipeline resources available to be filled with instructions than memory bandwidth to fully satisfy a single thread's request stream. Architectures such as Sun's Niagara 2 [70], Intel's Core i7 [7] and NVIDIA's Fermi Architecture [9] all use cores

that are multi-threaded in some fashion to increase throughput and better tolerate memory access latency.

Because of the increased amount of sharing between threads in an MT core, it is important to determine how to allocate the portion of available fetch bandwidth properly among threads to make use of the pipeline and cache resources effectively. Especially in the realm of transactional memory processors, which, if not managed properly can spend disproportionate amounts of resources executing useless code. Many researchers have looked at sharing pipeline resources. Prior work focused primarily on throughput of the system, and maximizing the number of instructions per cycle. This was the aim of Tullsen et al.'s work with the ICOUNT policy [107]. Increasing throughput though can be a poor metric if an application is latency sensitive and fairness is needed. Work by Raasch and Reinhardt [89] looked at the problem of fetch bandwidth allocation when latency and fairness is more important and found different policies other than ICOUNT were best. More recent work, such as work by Lakshminarayna and Kim [75] look at fetch policies for MT processors with the goal of limiting load imbalance in a GPU. By limiting load imbalance and giving priority to threads that need to complete work the most (but the thread may not be the most efficient thread) helps parallel scaling by limiting the number of idle cores waiting at a barrier for a slow thread, in turn increasing overall system throughput.

This chapter looks at parallel scaling and load imbalance in transactional applications running on an MT enabled core that also has Eager/Eager HTM support per thread in the MT core. As seen in past chapters the STAMP benchmarks have many bottlenecks that lead to poor performance. For an MT processor, it is important to devise TM aware fetch policies to avoid giving bandwidth to threads running code that is not providing forward progress. I propose in this chapter an instruction fetch policy that uses contention prediction to guide which threads to fetch for on a given cycle to maximize useful throughput. The following sections will present the implementation details and provide evaluation of this proposal.

6.2 Implementation

6.2.1 MT Architecture Model

The MT architecture modeled in this chapter is shown in Figure 6.1 and assumes a single pipeline, heavily threaded architecture similar to the Sun Niagara [71] processor. The architecture assumes the fetch unit is a shared piece of logic that picks which hardware context to fetch from each cycle. It assumes all the registers and transaction checkpoints are physically duplicated to support multiple hardware contexts. Other pieces of hardware

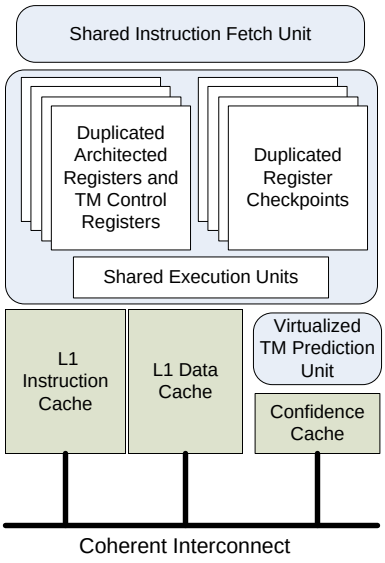


Figure 6.1: The modeled MT Architecture in M5

are virtualized and shared instead of being duplicated, like the thread prediction unit. The caches also need modification to allow each hardware thread to be executing a transaction and provide conflict detection in the shared L1 data-cache. This is done by having the cache perform checks for conflicting accesses in much the same way the coherence policy for the transactional system does. A transactional cache access is now a two step process, before the access to the cache array, the address must be compared to the signatures of the other threads on the core to detect local conflicts, then if the access proceeds, the coherence mechanism will perform conflict detection among the remote caches. As with any multi-threaded processor, it will be larger than a single threaded core, but is still smaller than duplicating cores to make a larger system.

The fetch unit on the MT-core uses a FAIR fetch policy similar to the fetch policy used in the work by Lakshminarayna and Kim [75]. The fetch unit decides which thread to pick for execution each cycle by picking the thread that has executed the least number of instructions. To implement this, the fetch unit uses counters for each hardware context to increment as instructions are fetched for the contexts. A comparator can be used to pick the thread context having executed the fewest number of instructions. When all the threads are getting roughly the same performance this fetch policy is approximately ROUND-ROBIN. To prevent pathological fetch cases where one context blocks on a very long latency operation and stops fetching instructions and on return uses all the available fetch bandwidth to catch up, the fetch unit uses defined epoch to determine when to zero the counters.

The primary reason the FAIR fetch policy was chosen is due to the ease in which it can

be modified to allow for bandwidth partitioning among threads. The intuition behind this is that the FAIR policy fetches based on the thread that has fetched the fewest number of instructions. To reallocate bandwidth among threads is simply a function of modifying by how much each counter increments for each instruction fetched and executed. A thread with more bandwidth allocated to it will count by incrementing the counters by 1, and threads that have less bandwidth will increment their instructions counters with a number >1. In this model, the cores perform bandwidth allocation in a distributed manner and it is assumed they all start with an equal division of fetch bandwidth. When a thread detects that it should not be using its full bandwidth allotment it signals the fetch unit to reduce its bandwidth allotment to some fraction of its original bandwidth (one half, one fourth, one eighth etc.). By extension this reduction in bandwidth for one thread is then equally redistributed among the remaining threads. For example, take a processor with two threads and each thread has half the bandwidth of the fetch unit (incrementing each counter by 1). If thread 1 wants to reduce its bandwidth by half and give it the other half to thread 0 it would start counting instructions by incrementing its counter by 2 instead. This gives thread 0 $^3/_4$ of the bandwidth and thread 1 $^1/_4$ of the fetch bandwidth. For the remainder of this chapter this is known as a *Bandwidth Modifier* of 2. A *Bandwidth Modifier* of 4 would have thread 1 increment its counter by 4 and have a resulting bandwidth of $^1/_8$. For cores with more threads, using the *Bandwidth Modifier* ends up splitting up bandwidth less effectively across threads as it is spread evenly to the other threads. This use of *Bandwidth Modifier* to modify the value of incrementing the instruction is a simple and implementable (needs only modify a register value by using a shift) way to proportion bandwidth among threads. Because bandwidth can be asymmetrically provisioned this can give useful threads a performance boost because they can fetch and execute more instructions and by extension can use more cache space implicitly. To decide when to re-provision bandwidth among threads to provide acceleration, threads detect when they are less useful and give up bandwidth. In the next section I will describe how this is easier to detect than detecting the candidate thread to boost via voltage boosting as in Chapter 5 for transactional applications.

6.2.2 Identifying When to Throttle

As seen in Chapter 5, it is a difficult problem to determine which thread is causing a scalability bottleneck and then boosting it to clear the bottleneck quickly. On top of this only about half of the tested benchmarks saw a sizable performance gain. It was also a hard problem due to the requirement of the boosting hardware. The hardware needed global visibility of the entire system or cluster to make boosting decisions. This hardware may or may not be realizable.

Benchmark	Input Parameters
Delaunay [72]	-i large.2 -m30 -t64
Genome	-g4096 -s32 -n524288 -t64
Kmeans	-m20 -n20 -t0.05 -i random50000_12 -p64
Vacation	-n8 -q10 -u80 -r65536 -t131072 -c64
Intruder	-a10 -l32 -n8192 -s1 -t64
Ssca2	-s15 -i1.0 -u1.0 -l3 -p3 -t64
Labyrinth	-i random-x96-y96-z3-n128.txt -t64
Yada	-i large.2 -m30 -t64

Table 6.1: STAMP Benchmark input parameters.

In the case of MT fetch throttling, it is an easier problem to detect when a thread is not executing useful code and therefore be given limited fetch bandwidth in the scope of transactional applications. The BFGTS scheduling algorithm makes a local decision about the thread attempting to run and whether it is profitable or not. These scheduling decisions made by the contention manager can be used to guide the fetch bandwidth provisioning. When a thread needs to be stalled, the BFGTS hardware automatically stalls the thread by assuming the thread being stalled behind has a short runtime. This automatically gives bandwidth to the remaining hardware contexts. If BFGTS determines a thread must serialize behind a large transaction it causes a `pthread_yield()` to occur to switch in another thread. This `pthread_yield()` is not contributing to the overall execution of the program, and can be safely slowed down by making the thread executing this function give up fetch bandwidth as per its *Bandwidth Modifier*. When the slowed thread again begins to execute transactions, its fetch bandwidth is restored. In the case of the STAMP benchmarks that execute transactionally almost all the time, this is an adequate design. It would not be hard to add additional hooks such as into the `schedule()` function of the Linux kernel to make additional decisions about fetch bandwidth allocation. As it can be seen, it is much easier to determine when to throttle a thread down than it is to determine when to throttle a thread up. In the next section I will provide evaluation of using fetch bandwidth to speed up execution of transactional applications that are running on multi-threaded processors.

6.3 Evaluation

6.3.1 Simulation Environment and Benchmarks

As in the previous chapters, the STAMP benchmarks are used to evaluate the idea of MT fetch throttling. The parameters used are listed above in Table 6.1. They are the same inputs as used in the previous chapters, and use 64 threads as in chapters 3 and chapter 4.

Feature	Description
Processors	2,4,8,16 one IPC Alpha cores @ 2GHz
Threads per Core	1,2,4,8 Threads per Alpha Core
Thread Fetch Policy	Fair fetch policy
Fetch Bandwidth Modifier	1, 2, 4, 8
L1 Caches	64kB-512kB, 1 cycle latency, 2-way associative, 64-byte line size
L2 Cache	32MB, 24 cycle latency, 16-way associative, 64-byte line size
Interconnect	Shared bus at 2GHz
Main Memory	2048MB, 100 cycles latency
Linux Kernel	Modified v2.6.18
Contention Managers	BFGTS-NoOverhead
Signature Size	Perfect signatures used for conflict detection and BFGTS-NoOverhead.

Table 6.2: M5 Simulation Parameters.

The M5 simulation parameters are listed in Table 6.2. I test using 2,4,8 and 16 cores using the BFGTS-NoOverhead contention manager to test only the effects of fetch bandwidth throttling. The number of hardware threads is kept constant at 16, so for a 4 core system it is 4-way multi-threaded. Also note that the L1 cache sizes are varied to see what the effects of limiting the cache has on the performance, from using only 64kB of cache per core to using 64kB of cache per thread. The *Bandwidth Modifiers* tested range from no bandwidth modification when detecting throttling is needed (value of 1) to a *Bandwidth Modifier* of 8.

6.3.2 Performance and Sensitivity Analysis

In this evaluation I present each benchmark individually and provide analysis of its overall performance as well as provide insight into its sensitivity to: cache size, bandwidth modifier value, and core count. The plots that are presented in this benchmark are per benchmark and are read as follows: The y-axis is speedup compared to one core execution. The x-axis is divided into configuration groups. From right to left the configuration groups are the following: 16 cores with one thread per core, 64kB of cache per core. 8 cores with 2 threads per core, bandwidth modifiers from 1-8 and cache varying from 32kB-64kB per thread. 4 cores with 4 threads per core, bandwidth modifiers from 1-8 and cache varying from 16kB-64kB per thread. Finally 2 cores with 8 threads per core, bandwidth modifiers from 1-8 and cache varying from 8kB-64kB per thread. The lines show the trend of varying the bandwidth modifiers.

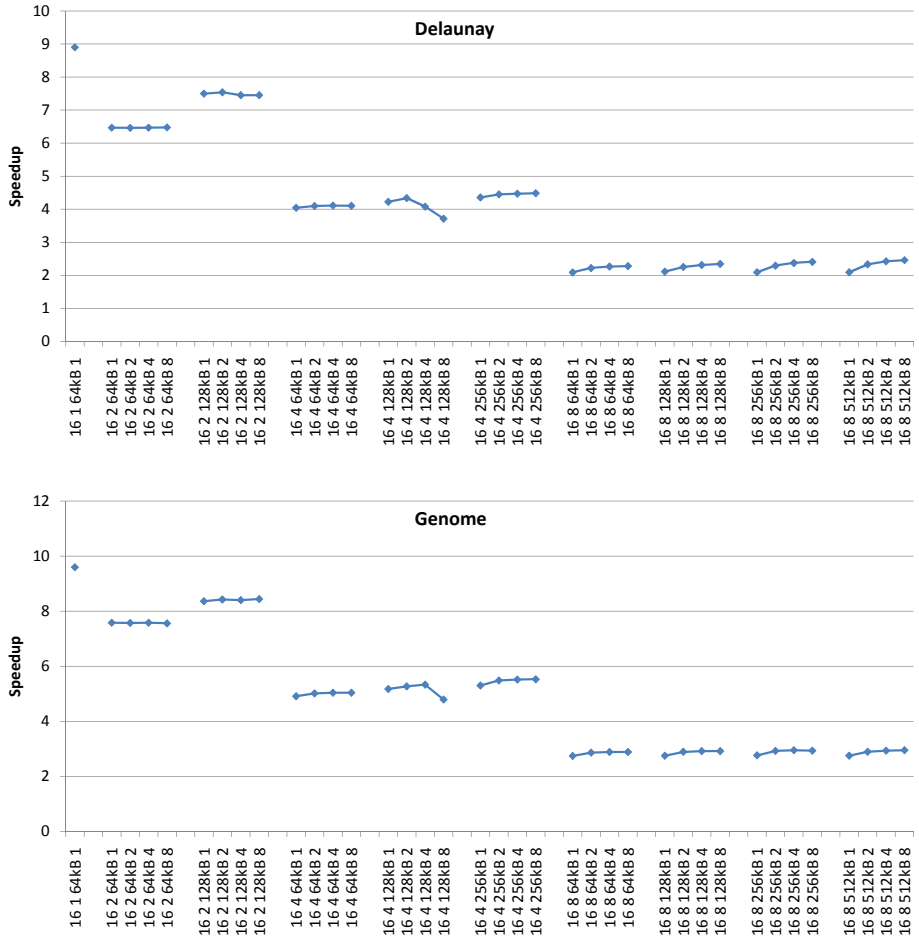


Figure 6.2: Performance and sensitivity to number of threads, L1 cache size and bandwidth allocation for the Delaunay and Genome STAMP Benchmarks.

6.3.2.1 Delaunay

Figure 6.2 shows the performance results of the *Delaunay* and *Genome* benchmarks when using MT fetch throttling. For the *Delaunay* benchmark, the performance steadily decreases as the number of physical cores decrease. The 16 core configuration gets the best performance as it should. The *Delaunay* benchmark as shown in previous chapters has a fair amount of contention and scalability bottlenecks to potentially be accelerated. Interestingly enough, with the 4 and 8-way multi-threaded cores (4 and 2 physical cores respectively), the system can actually get performance greater than the number of cores over the single core baseline. This is because larger number of threads are able to execute during long latency misses and some pre-fetching benefits are seen with the multi-threaded cores for this application. As the cache is increased per thread, the 2 and 4-way threaded cores show

some sensitivity to cache size. The 2-way threaded version with 8 cores sees the most gain. The sensitivity to the *Bandwidth Modifier* shows interesting behavior. For the 2-way multi-threaded cores, reassigning bandwidth to different threads has little impact. This is due to the small amount of contention for resources between 2 threads per core. When moving to 4-way multi-threading, the application shows some sensitivity as threads give up bandwidth to accelerate others. Because there is more competition for cache bandwidth, this has an affect on performance, gaining a few percent. The results for the 4-way configuration even shows that giving up too much bandwidth can result in performance degradation as in the case for the *Bandwidth Modifiers* 4 and 8 for one of the 4-way configurations. When the system is 8-way multi-threaded there is a discernible advantage to using fetch throttling to improve performance. In this case a *Bandwidth Modifier* of 8 is the best to reduce contention on the scarce pipeline and cache resources in an 8-way multi-threaded processor. In the 8-way multi-threaded configuration fetch throttling can get up to 17% performance improvement using a *Bandwidth Modification* of greater than 1.

6.3.2.2 Genome

The *Genome* benchmark in Figure 6.2 shows different behavior from *Delaunay* benchmark presented above. As seen in the previous chapter, *Genome* has a greater amount of parallelism and does not benefit from boosting techniques. As the number of cores decreases, performance steadily decreases as is seen in *Delaunay*. Again though, the performance over one core is still higher than the number of physical cores as again we are seeing the utilization and prefetching benefits of multi-threading. Increasing the amount of cache per thread does not have as large of an impact as it did in the *Delaunay* benchmark. This makes sense as the *Genome* benchmark has smaller transactions on average, and caching performance is not quite as critical. Because *Genome* has fewer scalability bottlenecks than *Delaunay* it sees less benefit from the *Bandwidth Modifier*. Still, there is a case again where reducing bandwidth too much can lead to reduced performance for a modifier of 8 in the 4-way multi-threaded configuration. Overall, reducing bandwidth can get up to a 7% increase in performance for the most resource constrained 8-way multi-threaded configuration.

6.3.2.3 Kmeans

Figure 6.3 shows the performance and sensitivity to the different parameters for the *Kmeans* and *Vacation* benchmarks. The *Kmeans* benchmark is a very parallel benchmark that sees almost linear scaling for the 16 core configuration. As expected it too sees a performance degradation as the number of physical cores is reduced and the number of

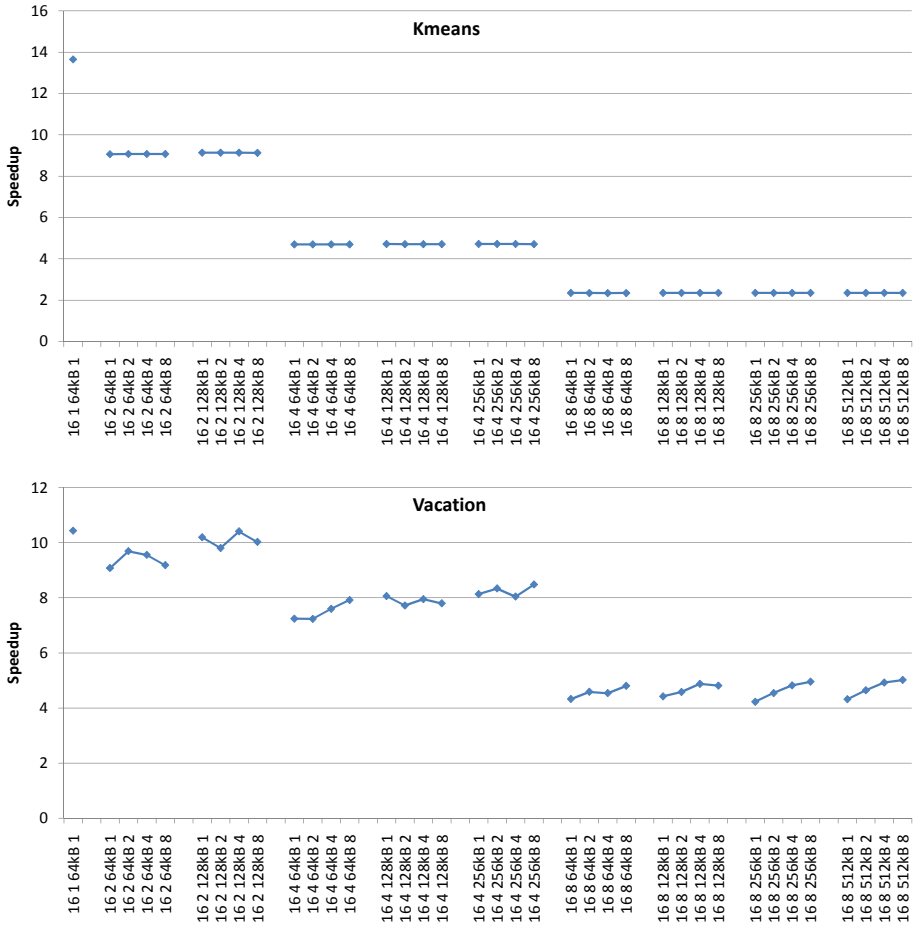


Figure 6.3: Performance and sensitivity to number of threads, L1 cache size and bandwidth allocation for the Kmeans and Vacation STAMP Benchmarks.

threads is increased per core. Again each multi-threaded configuration gets better speedup than the physical number of cores available as expected. It is insensitive to cache per thread as all the transactions executed by *Kmeans* are very small. Because the *Kmeans* benchmark is very parallel with no real bottlenecks to be accelerated it is also insensitive to the *Bandwidth Modifier* parameter, seeing no appreciable gain.

6.3.2.4 Vacation

The *Vacation* benchmark is another benchmark that has some bottlenecks that can be accelerated by using the *Bandwidth Modifier* as well as other interesting properties. In this case the 16 core configuration and the 8 core 2-way multi-threaded configuration with 64kB cache per thread achieve the same performance. The 4-core 4-way and 2-core 8-way multi-threaded get very good performance as well, about 2x the performance to be

expected from the number of physical cores available. The *Vacation* benchmark works on one large shared Red-Black tree data-structure during its execution. The performance seen from the multi-threaded processors suggests that the *Vacation* benchmark is benefitting from the utilization and pre-fetching effects seen from multi-threading and there is very little cache pollution among the threads. In the case of cache sensitivity, the 2-way and 4-way multi-threaded cores are sensitive to cache space per thread, while the 8-way multi-threaded core is less sensitive to cache per thread. All the multi-threaded configurations are very sensitive to the *Bandwidth Modifier* parameter for fetch throttling. The 2-way multi-threaded cores show a modifier between 2 and 4 as being the best, while 8 leads to a performance degradation. For the 4 and 8-way multi-threaded cores, a *Bandwidth Modifier* of 8 is the best overall. This is due to the restricted resources on these configurations and the extra fetch bandwidth allowed by the high modifier to useful threads allows them to get to use more cache space and complete transactions faster so serializing transactions get to run sooner than they otherwise would. For *Vacation* an up to 17% performance can be seen using fetch throttling for aggressive 8-way multi-threading. Up to 10% can be seen for a less aggressive 4-way multi-threading configuration when using a *Bandwidth Modifier* greater than 1.

6.3.2.5 Intruder

Figure 6.4 shows the performance and sensitivity to the test parameters for the *Intruder* and *Ssca2* benchmarks. The *Intruder* benchmark has a very high contention rate as seen in the previous chapters and therefore has many scalability bottlenecks that could be accelerated as seen in chapter 5. The *Intruder* benchmark saw the greatest benefit of all benchmarks tested from voltage boosting. It also sees the greatest benefit to fetch throttling for multi-threaded processors. Because of the high amount of contention limiting performance for *Intruder*, the 16 processor configuration and 2 processor 8-way multi-threaded configuration using fetch throttling can get almost equivalent performance. Overall this benchmark can be considered insensitive to multi-threading as it has very little parallel potential past a 2.5x speedup. *Intruder* does see sensitivity to cache space per thread for all the multi-threaded configuration, seeing gains in performance as cache is increased with the max performance being attained for 64kB of cache per thread. In terms of sensitivity to fetch-throttling the *Intruder* benchmark is very sensitive. It sees the greatest gains for a *Bandwidth Modifier* of 8 in all cases. This is due to the numerous bottlenecks experienced by the *Intruder* benchmark and boosting performance of threads doing useful work sees large gains. Overall using fetch throttling for multi-threaded processors can see up to a 213% percent performance improvement for the 8-way multi-threaded configuration over

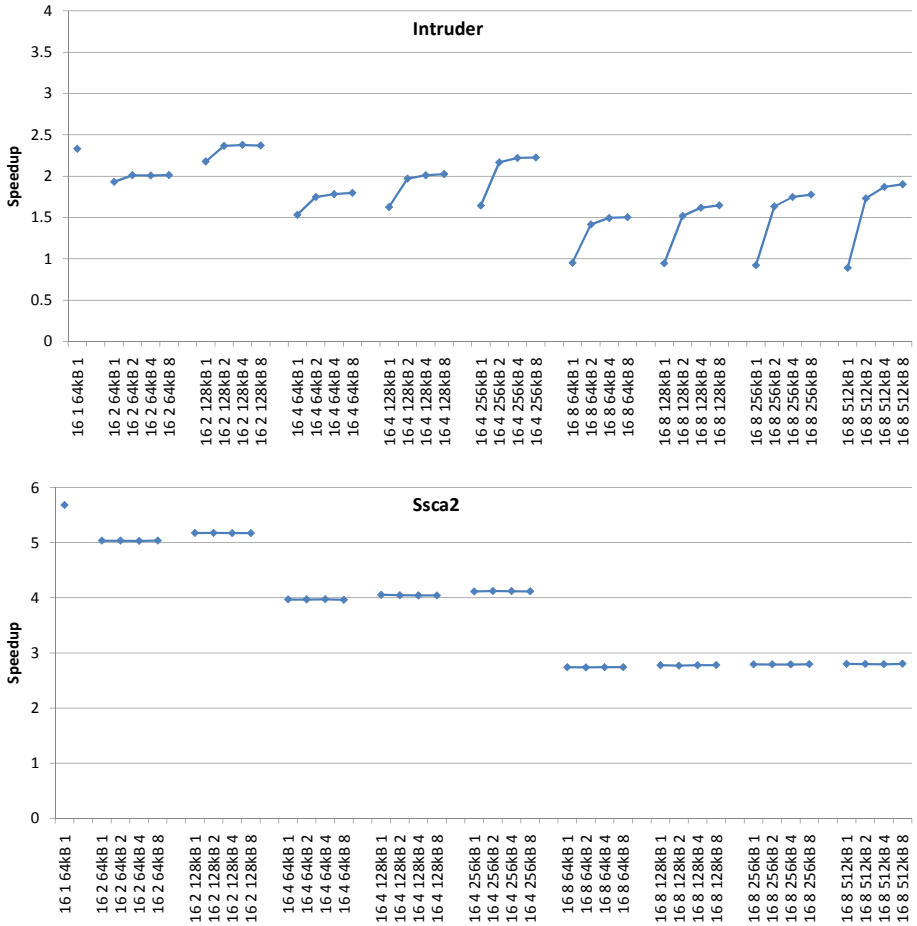


Figure 6.4: Performance and sensitivity to number of threads, L1 cache size and bandwidth allocation for the Intruder and Ssca2 STAMP Benchmarks.

using no fetch throttling and giving all threads equal bandwidth. For the 2-way and 4-way multi-threaded configurations the gains are less but still significant getting a maximum of 17% and 57% performance gains respectively over a bandwidth modification of 1.

6.3.2.6 Ssca2

As seen in Figure 6.4 the *Ssca2* benchmark exhibits almost identical behavior to the *Kmeans* benchmark as seen in Figure 6.3. Because of poor caching behavior *Ssca2*, it does not scale linearly, but it should due to its very low (<1%) contention as seen in previous chapters. Because of this there are no bottlenecks to be accelerated and the 16 processor configuration gets the best performance. The multi-threaded configurations get slightly better performance than the number of physical cores due to the utilization and pre-fetching effects as seen with the previously analyzed benchmarks. Because of *Ssca2*'s poor caching

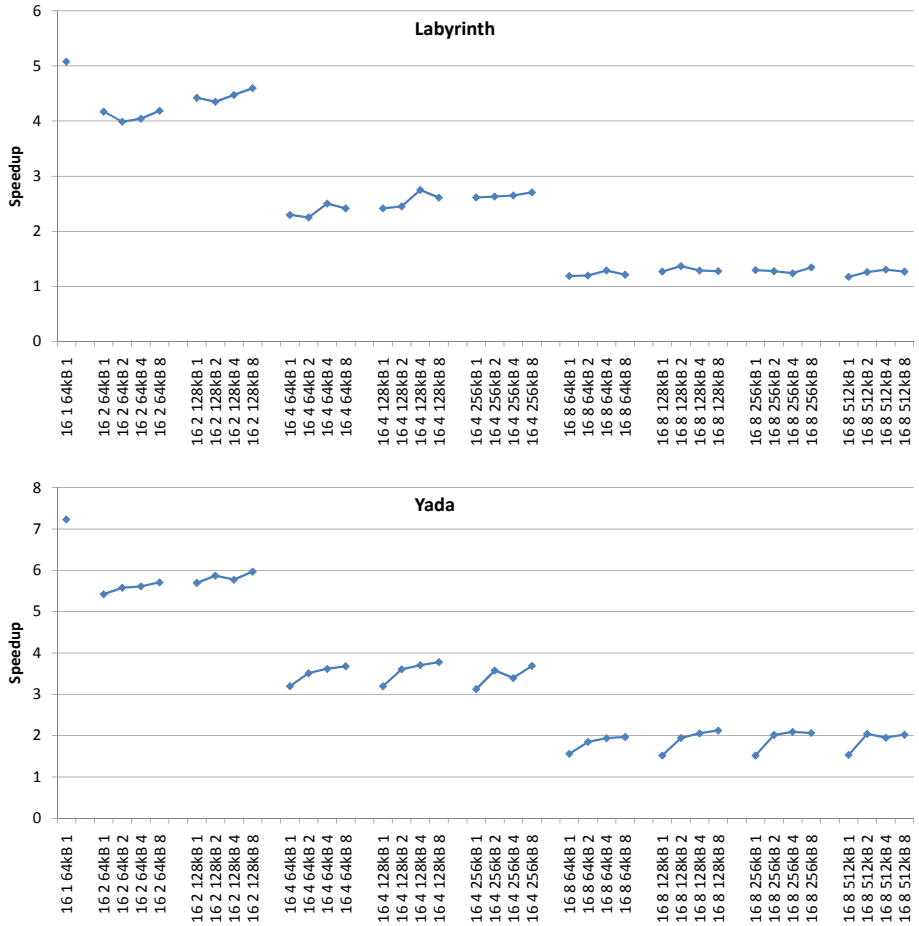


Figure 6.5: Performance and sensitivity to number of threads, L1 cache size and bandwidth allocation for the Labyrinth and Yada STAMP Benchmarks.

characteristics it does have sensitivity to cache per thread. *Ssca2* sees definite improvement in performance as cache per thread is increased. As mentioned before, the lack of any bottlenecks means that fetch throttling attains no performance gains for this benchmark as shown in the results.

6.3.2.7 Labyrinth

Figure 6.5 shows the performance and sensitivity of the *Labyrinth* and *Yada* benchmarks. The *Labyrinth* benchmark exhibits behavior similar to what has been seen in the previous chapters, namely random behavior as it is a very hard benchmark to apply scheduling and boosting predictions to it effectively. It does not show much scaling and multithreading actually degrades performance as the physical cores decrease and threads are added. It does show some sensitivity to cache size per thread as seen in Figure 6.5, this is

due to the *Labyrinth* benchmark having long transactions with large read/write sets. The *Bandwidth Modifier* parameter shows some sensitivity but there is no clear trend exhibited as to which setting of *Bandwidth Modifier* is the best. These results are expected due to *Labyrinth's* random behavior seen in previous chapters.

6.3.2.8 Yada

The *Yada* benchmark on the other hand as seen in Figure 6.5 has clear performance trends and sensitivities. As the number of physical cores decrease the performance also decreases as would be expected. In the case of *Yada* it does not benefit from any pre-fetching or increased utilization effects of a multi-threading core as seen by the lower than physical core count speedups. *Yada* does on the other hand see some benefit to increasing the amount of cache per thread for the 2-way multi-threaded configuration. For the 4-way and 8-way multi-threaded configurations extra cache space per thread appears to have little effect. Throttling fetch bandwidth via the *Bandwidth Modifier* has a large effect on the *Yada* benchmark. For all the configurations, a *Bandwidth Modifier* of 8 has a large effect on performance, getting up to a 39% performance increase over using no fetch throttling when using an 8-way multi-threaded core. For 2-way and 4-way, a *Bandwidth Modifier* of 8 sees a 5% and 18% performance increase which is significant given these configurations lower contention for shared resources.

6.4 Conclusions

The results presented in this chapter show BFGTS scheduling decisions can be used to guide fetch policy decisions for multi-threaded cores. This opens up many future avenues of research that could be investigated. One is to look into methods to change which bandwidth modifier to use for each benchmark. As shown in the results, no one bandwidth modifier value performed best overall. Another area to research would be to find different methods for assigning bandwidth to threads and enforcing the bandwidth limitations. The current method is very simple, and appears to work well, but may not be optimal. Finally it may be informative to use of a more detailed CPU model and evaluate if the fetch policy still works as seen here for simple 1 IPC cpu models.

As with voltage boosting, throttling fetch bandwidth in a multi-threaded core can improve performance overall in a multi-threaded architecture when running applications that exhibit scalability bottlenecks. As shown here the task of identifying when to relinquish fetch bandwidth from one thread to other threads to allow faster execution is a substantially easier a problem than determining when to boost a core by looking for potential bottle-

necks. Its other advantage is this bandwidth throttling is completely distributed and the decision can be made locally, negating the need for centralized hardware like in Chapter 5. Because of these reasons, almost all benchmarks saw an improvement in execution time for sufficiently resource constrained multi-threaded architectures (4 and 8-way multi-threaded cores) when using fetch throttling except for two benchmarks that exhibited few or no bottlenecks. Overall this technique of fetch throttling saw upwards of a 200% performance improvement over FAIR scheduling of instructions for transactional applications using a scheduling contention manager to guide fetch bandwidth throttling decisions.

CHAPTER 7

Conclusions

The power and complexity limitations placed on architects for designing single-threaded cores has led to shifting focus from the traditional powerful single-core processor to the chip multi-processor (CMP). Because of this shift to CMPs, many areas of parallel programming need to be made tractable to all programmers and not just a small group of experts. One area that has received a large amount of attention is making synchronization on shared memory CMPs accessible to programmers other than experts. Transactional Programming through the use of Hardware Transactional Memory has been proposed to make synchronization of critical sections more tractable.

But, Hardware Transactional Memories (HTMs) experience problems with scalability bottlenecks caused by less than optimal construction of parallel programs using transactions. Transactional programs written in a fashion that emulate synchronization patterns that would be used by less experienced programmers—a small number of large critical sections—experience scalability bottlenecks that can cause severe performance degradation (performance can be less than serial exection) when executed on HTMs. This performance problem is unacceptable for these systems as it can not be expected of the programmer to tune transactional programs to increase performance. If tuning was required, all the programmability benefits of transactional programming would be lost. Therefore the underlying HTM that promises fast execution must provide facilities to automatically resolve these scalability bottlenecks.

In this thesis I developed and evaluated a hardware/software approach that specifically leverages the properties of HTM to eliminate scalability bottlenecks without programmer intervention for transactional programs. HTM specifically allows dynamic profiling of critical section dependencies because the HTM system exposes critical sections to the hardware. This is not easily achievable with locks as they are not associated with critical sections in a way that is exposed to the system to use and reason about effectively. Multiple

techniques were developed to dynamically, at run-time, tune the execution of an HTM system to alleviate scalability bottlenecks.

First I proposed that proper contention management design can eliminate scalability bottlenecks by using a technique called ”Proactive Transaction Scheduling” (PTS). PTS constructs a dynamic conflict graph and predicts future conflicts to avoid future contention and increase performance. PTS also made the observation that an over-committed system (more threads than cores) could be leveraged to hide serialization latency between critical sections by forcing the operating system to schedule different threads that could contain independent work, making better use of the multi-core system. PTS attained a 2x greater performance on average over a reactive backoff based contention manager, and was accomplished completely as a software runtime.

Enhancements to the PTS contention manager called ”Bloom Filter Guided Transaction Scheduling” (BFGTS) was then developed. BFGTS made the observation that Bloom filters could be used to better infer transaction behavior and better guide scheduling decisions. Specifically Bloom filters were used to measure “Similarity”, a metric that characterized a transaction’s historical read/write set behavior in terms of randomness. “Similarity” was then used to guide when to treat a transaction optimistically (schedule to run concurrently) or pessimistically (serialize behind other transactions) more accurately than could be attained with PTS. BFGTS attained a 21% average performance improvement over PTS and did not need special application specific optimizations as were required in PTS. BFGTS also attained a 31% average performance improvement over an outside competing technique called “Adaptive Transaction Scheduling” (ATS). BFGTS used a combination of software runtime and small hardware additions to attain these improvements.

I then applied the insights gained from developing PTS and BFGTS to dynamically profile and re-order execution of transactions in regards to Asymmetric Multi-processing (AMP). AMPs allocate processing resources asymmetrically among cores to allow some threads access to high single thread performance if needed. One of the main problems with AMPs is assigning threads to the correct cores to attain optimal runtime. Using work by Dreslinski et al. [51, 52] that establishes voltage boosting can be used effectively to make an AMP, I applied the transaction scheduling techniques developed previously to determine the transaction causing scalability bottlenecks and directing the hardware to voltage boost that thread. This led to a 15% average performance improvement.

Finally I showed that transaction scheduling could also be applied to Multi-Threaded (MT) processor fetch policy. The main problem in this domain is to identify when threads were needlessly consuming fetch bandwidth and execution resources. These wasteful threads were characterized by executing code that was not contributing to the overall pro-

gram execution (i.e. spinning on a lock, context switching etc). I showed that transaction scheduling could be used to identify threads that were not contributing to the completion of the program. By identifying these threads, fetch bandwidth could be repartitioned to threads that were making useful progress. I show that on heavily multi-threaded architectures properly throttling fetch bandwidth among threads could lead up to a 2x improvement in overall performance.

7.1 Future Work

This thesis has shown that transactional memory allows for interesting solutions to scalability bottlenecks. In particular it allows for dynamic profiling at runtime that can be leveraged to arrive at better schedules of execution for threads and make effective use of AMPs. TM allows this because it exposes the concept of a critical section to the underlying system in a fashion that allows the system to see the interactions among critical sections and how they evolve. The exposure of critical section behavior allowed the development of the transaction scheduling runtimes presented, exposing other program behavior properties will need to be researched to allow development of better runtimes to leverage larger future CMPs.

As systems continue to get larger in terms of processing cores, and heterogeneity increases due to power and performance concerns, online profiling and targeted system management will become more critical. This is especially important in the area of scheduling on AMP systems. In the context of this thesis, it was guiding when and which core to voltage boost to gain single-threaded performance. Work will need to be continued in this area to properly leverage these future systems.

There are also outside uses for the "Similarity" metric developed in this thesis that could be investigated in the future. One could be applying it to cache sharing among processes in a CMP to maximize throughput by using it to identify the working sets of a thread in a more compact manner. Another could be to potentially apply "Similarity" to the memory stream to provide quality of service on the memory interconnect.

Lastly, the concept of transaction scheduling is still receiving attention in the TM community. Future work could be done to further improve the PTS and BFGTS schedulers. As seen in Chapters 3 and 4 there is still a fair amount of predictions that are predicting to serialize or parallelize incorrectly. Improving the predictions further will allowed for increased performance.

Overall this thesis provides techniques and insights that can be applied to future research in the area of runtimes for dynamic tuning to improve performance and utilization

of large CMPs. This research can be in both the areas of the runtime itself as well as what aspects of program behavior should explicitly be exposed to the architecture to allow runtimes to tune execution better.

APPENDIX

APPENDIX A

Additional Experiments

This thesis presented many techniques related to transaction and thread scheduling. In all but Chapter 5, all the configurations evaluated were over-committed with more threads than cores. This allowed the scheduling algorithms to hide serialization latency and therefore gain performance. This appendix presents experiments conducted in Chapters 3 and 4 using the same number of threads as cores using the STAMP benchmarks instead of overcommitting the system.

A.1 Backoff Non-Overcommitted System Results

As seen in Tables 1.1 and 1.2, the STAMP benchmarks suffer from high contention when the processor count reaches 16 cores and in turn sees limited performance scaling. That table was for an over-committed system using 64 threads. Tables A.1 and A.2 show the same experiments for a randomized backoff contention manager when the system is not over-committed—threads equal to cores.

Tables A.1 and A.2 show the results are very similar. In all cases except *Ssca2* the STAMP benchmarks behave almost identically whether the number of threads is greater than or equal to the number of cores. Speedups and contention values are very similar. *Genome* and *Intruder* still see pathological contention that leads to less than sequential performance. *Delaunay* and *Yada* still see limited scaling due to their large amount of measured contention. *Ssca2's* performance sees a marked improvement. This is due to its data-structures being better aligned in the cache from having less threads allocated. Still *Ssca2* has less than linear speedup, suggesting that its cache performance is still not particularly good.

	Delaunay	Genome	Kmeans	Vacation
2	1.3	1.9	1.9	1.9
4	2.0	3.4	3.9	3.8
8	3.0	4.3	6.7	4.8
16	3.5	0.9	6.7	6.2
	Intruder	SSCA2	Labyrinth	Yada
2	1.5	1.9	1.6	1.3
4	1.4	3.6	2.6	2.0
8	0.8	6.5	3.5	2.8
16	0.3	10.5	5.1	3.4

Table A.1: Speedup observed for STAMP Benchmarks with simple Randomized Backoff contention manager for a 2-16 processor system using a LogTM type Transactional Memory with threads equal to processors.

	Delaunay	Genome	Kmeans	Vacation
2	29.8%	0.3%	0.1%	0.1%
4	45.6%	1.8%	0.1%	0.5%
8	57.8%	8.8%	3.8%	4.0%
16	73.3%	65.1%	20.5%	13.2%
	Intruder	SSCA2	Labyrinth	Yada
2	2.5%	<0.1%	<0.1%	19.2%
4	12.8%	<0.1%	<0.1%	35.5%
8	39.0%	<0.1%	1.3%	42.9%
16	69.8%	<0.1%	15.5%	55.5%

Table A.2: Contention observed for STAMP Benchmarks with simple Randomized Backoff contention manager for a 2-16 processor system using a LogTM type Transactional Memory with threads equal to processors.

A.2 Reactive Thread-Yield Contention Manager Experiments

One of the base ideas of this thesis is: using an over-committed system is beneficial because other threads can be executed in place of threads that need to serialize in transactional memory applications. To apply this idea I created two rather complex transaction scheduling contention managers to take advantage of this insight. In this section I present experiments for a very simple reactive contention manager that tries to take advantage of an overcommitted system better than linear randomized backoff.

I designed a reactive contention manager that tries to backoff for a pre-defined period of time, and if the backoff cannot clear the conflict it initiates a `pthread_yield()` to try executing another thread. This contention manager is called the "Reactive Thread-Yield"

Benchmark	Speedup	Contention
Delaunay	1.2	92.6%
Genome	N/A	>99.9%
Kmeans	5.5	27.5%
Vacation	1.4	81.8%
Intruder	N/A	>99.9%
Ssca2	5.7	<0.1%
Labyrinth	5.3	10.7%
Yada	6.0	40.9%

Table A.3: Speedup and Contention observed for a Reactive Thread-Yield contention manager for a 16 processor LogTM type system using 64 threads.

contention manager. It was evaluated at 16 processors using my base LogTM implementation in the M5 Full System simulator using the setup parameters seen in Table 3.3 and evaluated on STAMP using the parameters from Table 3.2.

Table A.3 shows the speedup and contention results of the "Reactive Thread-Yield" contention manager. As seen in the table, the results are substantially worse than even the linear randomized backoff contention manger. Contention is much higher for the *Delaunay*, *Genome*, *Kmeans*, *Vacation* and *Intruder* benchmarks and performance is substantially worse. In the case of *Genome* and *Intruder* the benchmarks never finished due to pathological live-lock that the contention manager could not resolve with either backoff or thread yielding. *Ssca2* performs the same as backoff due to its low contention. *Labyrinth* also performs the same as backoff, as seen throughout this thesis the *Labyrinth* benchmark is contention manager insensitive. *Yada* is the only benchmark that sees a performance improvement. This improvement comes from the optimistic predictions of a reactive manager, and also gains benefit from switching in new threads to perform independent work. Overall though, the "Reactive Thread-Yield" contention manager is the worst performing manager of all those tested in this thesis and again shows that reactive managers are not effective when contention is high.

A.3 PTS Non-Overcommitted System Results

Chapter 3 presented all of its results using an overcommitted system. This section provides the results of applying PTS and PTS-Backoff to a non-overcommitted system. The STAMP benchmarks were run with the same parameters, except that threads are equal to the number of cores (16), and stalling is used in place of `pthread_yield()`. The analysis provided in this appendix is not as detailed as provided elsewhere due to these experiments

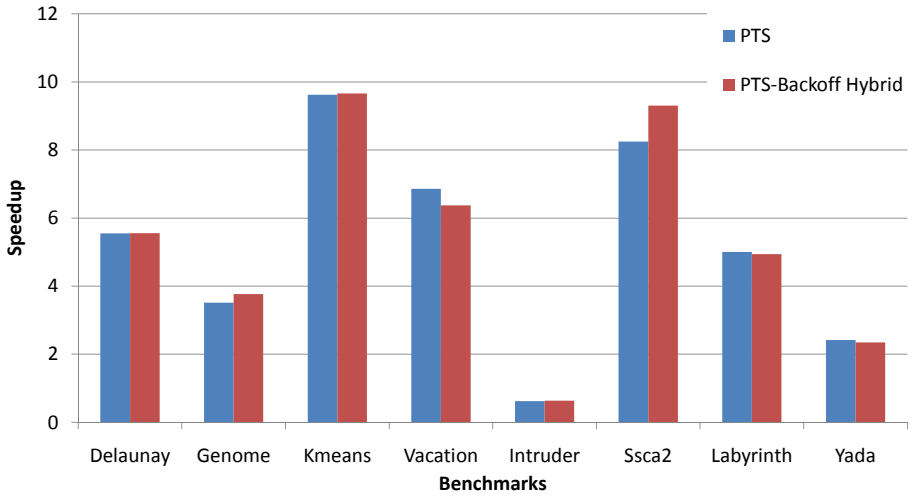


Figure A.1: Overall best speedup for PTS and PTS-Backoff for a 16 processor system using 16 threads.

Benchmark	PTS	PTS-Backoff
Delaunay	35.1%	35.3%
Genome	1.5%	1.1%
Kmeans	3.9%	3.2%
Vacation	4.5%	5.2%
Intruder	65.9%	65.3%
Ssca2	<0.1%	<0.1%
Labyrinth	21.8%	29.6%
Yada	13.6%	21.9%

Table A.4: Contention for PTS and PTS-Backoff for a 16 processor system with 16 threads.

being an additional data point for interested readers to compare to.

Figure A.1 and Table A.4 show the overall best performance and resulting measured contention for the STAMP benchmarks when number of threads is equal to cores. The main takeaway that should be noted is that the system slows significantly in all cases except for *Ssca2*. This slowdown is due to the system not being able to hide serialization latency any longer and bring in independent work. Resources are forced to go idle when a conflict is predicted. It should also be noted that contention also goes up when the system has equal number of threads to cores. In the case of *Intruder*, the contention goes up, and performance degrades, though not quite to the pathological nature of backoff based systems. On the other hand, *Yada* sees a decrease in contention along with large decrease in performance. This is due to PTS being overly pessimistic and serializing more than needed. This data shows that an overcommitted system is beneficial towards getting better performance in general.

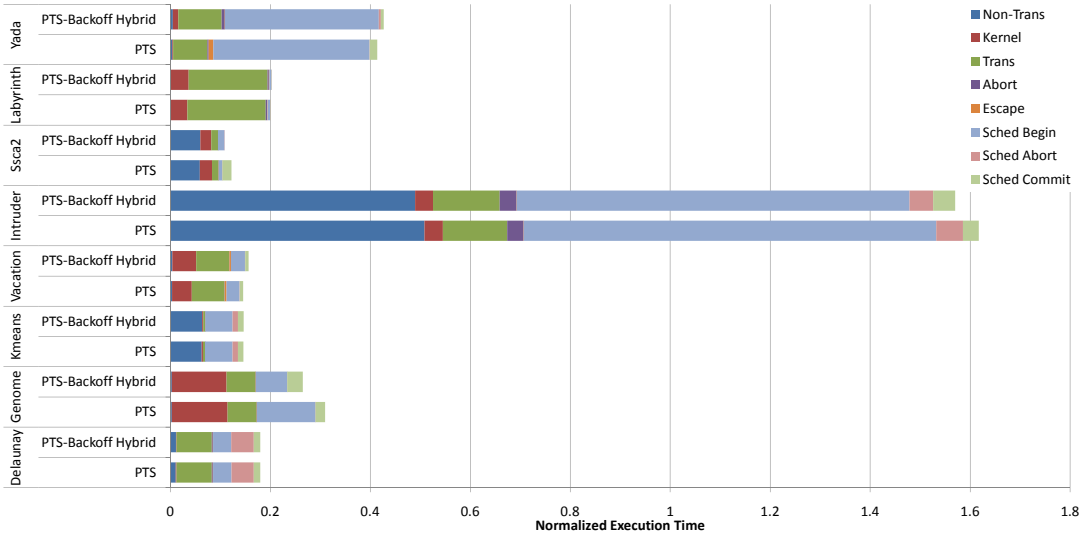


Figure A.2: Time breakdown of where PTS and PTS-Backoff spend time executing each benchmark normalized to the runtime of 1 core for each benchmark using a 16 processor system using 16 threads.

In the case of *Ssca2*, it gets better performance due to fewer threads taking up less memory and improving caching properties.

Figures A.2 and A.3 show the time breakdown and time distribution of where time is spent in PTS: Non-transactional, kernel, transactional, abort, escape, schedule begin, schedule abort and schedule commit. Compared to the overcommitted system results, more time is being spent in the "Sched Begin" and "Sched Abort" execution modes. For example, the *Yada* benchmark spends a large amount of time in "Sched Begin" because it is over serializing by making transactions wait in spin loops. Less time is spent in "Sched Commit" because while there are more serializations happening, less predictions are being made due to the lack of threads, meaning less Bloom filter compares are happening. This re-distribution of time spent in "Sched Begin" and "Sched Commit" help explain why the benchmarks are running slower, the predictions are not as good because less feedback is being provided from fewer predictions.

In the case of *Vacation* and *Genome* a fair amount of time is being spent in the kernel. While there are no calls to `pthread_yield()` to force entry into the kernel scheduler, load imbalance is possible where threads are hitting their barrier points much earlier than other threads leading to a large amount of time being spent in the kernel idle loop. Again this shows that an overcommitted system is also potentially easier to load balance, as the experiments in chapter 3 did not show large amounts of time being spent in the kernel.

Figure A.4 shows the sensitivity of PTS and PTS-Backoff to the "Small Transaction"

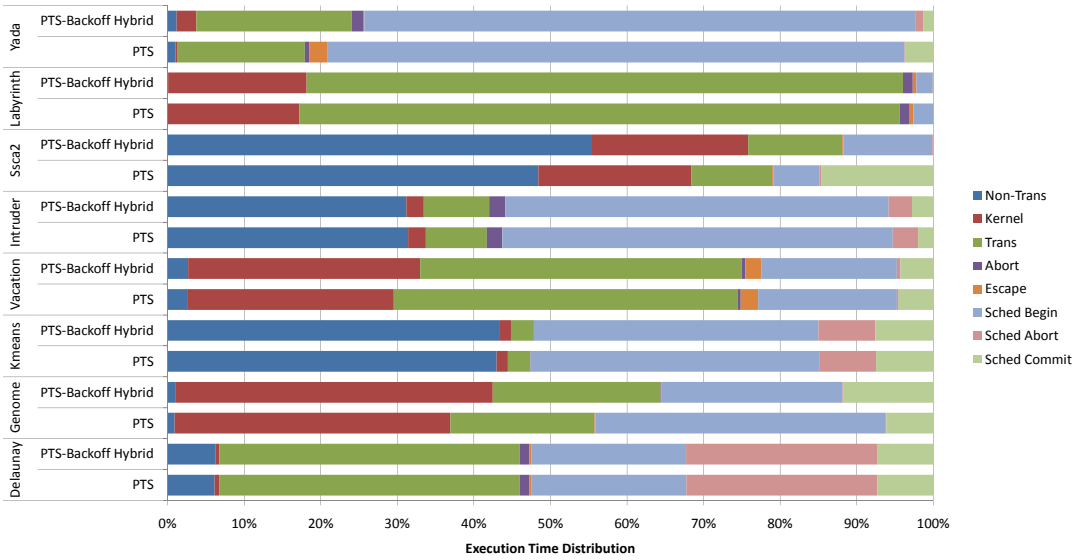


Figure A.3: Time distribution of PTS and PTS-Backoff executing each benchmark using a 16 processor system using 16 threads.

optimization when the system is not overcommitted. As can be seen from the figure, trends are almost identical to the results presented in chapter 3.

Figure A.5 shows the sensitivity of PTS and PTS-Backoff to the "Split Transaction" optimization when the system in not overcommitted. For the *Delaunay* and *Vacation* benchmarks, the trends are the same, there are improvements in performance. For the *Yada* benchmark, the trend does not hold as there is no improvement using or not using the "Split Transaction" optimization. The *Yada* benchmark has very pessimistic predictions regardless of optimizations made.

Figures A.6 and A.7 shows the sensitivity of PTS and PTS-Backoff to Bloom filter size for a non-overcommitted system. Again the trends are very similar to the results presented in chapter 3.

The major difference between running the system overcommitted or non-overcommitted is the benefit gained from having more threads than cores to load balance, and increase system throughput by scheduling independent work. When taking away this option, scheduling does have some benefit as seen by most benchmarks gaining performance over backoff, but the benefit is not as substantial.

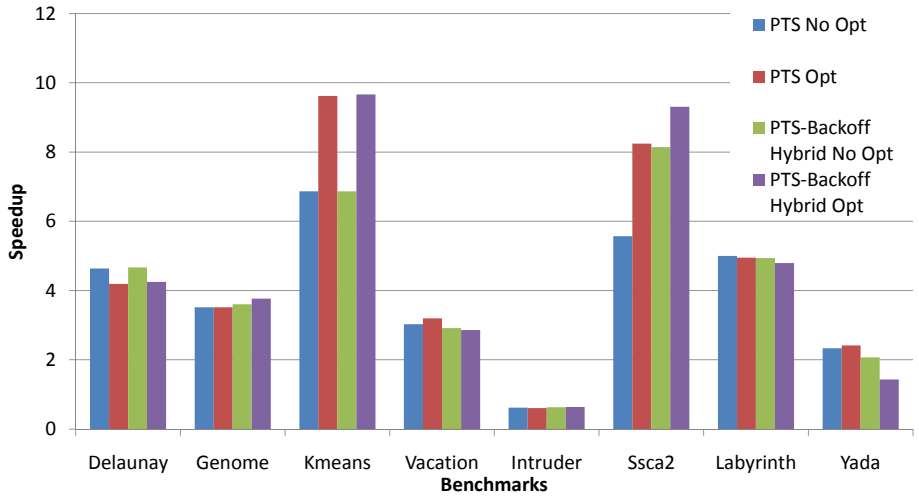


Figure A.4: Sensitivity of PTS and PTS-Backoff to the Small Transaction Optimization for a 16 processor system using 16 threads.

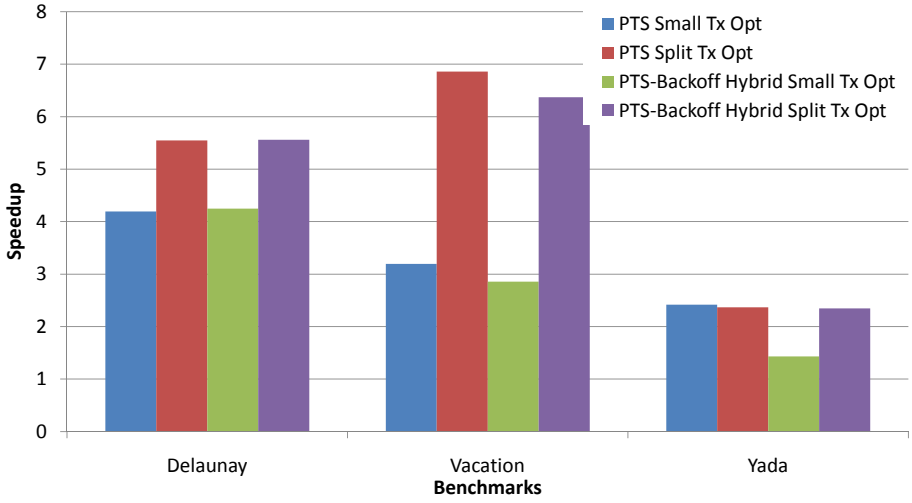


Figure A.5: Sensitivity of PTS and PTS-Backoff to the Split Transaction Optimization for a 16 processor system using 16 threads.

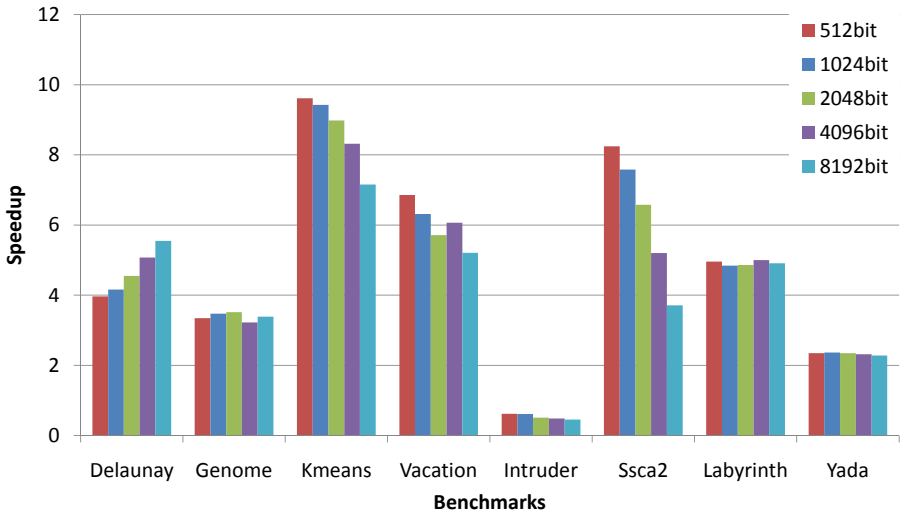


Figure A.6: Sensitivity of PTS to the Bloom filter size for a 16 processor system using 16 threads.

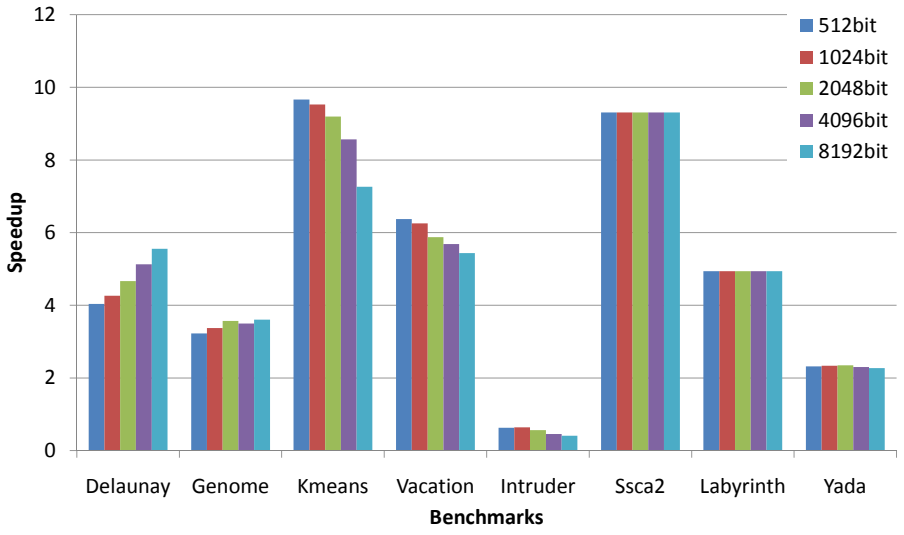


Figure A.7: Sensitivity of PTS-Backoff to the Bloom filter size for a 16 processor system using 16 threads.

A.4 BFGTS and ATS Non-Overcommitted System Results

This section presents non-overcommitted system results for BFGTS and ATS. BFGTS, as with the non-overcommitted PTS experiments stalls alls transactions to serialize them behind a predicted conflict as `pthread_yield()` cannot be used in this configuration. Because the system is non-overcommitted I also show an optimized version of ATS that uses spinlocks instead of pthread mutexes and condition variables to save on overhead. Spinlocks can only be used when the system is not overcommitted because there is no chance of a thread being pre-empted by the OS for an extended period of time. In any realistic system, there are always more threads than cores and spinlocks are not a viable option because if a thread is pre-empted while holding the spinlock all over threads will block while busy waiting. This section presents results for the optimal ATS configuration compared to BFGTS.

Figure A.8 and Tables A.5 and A.6 show the best overall speedup attained by BFGTS and ATS for a non-overcommitted system as well as the contention for both techniques. One should note that BFGTS attains on average much better performance than PTS when comparing Figure A.1 to Figure A.8 due to its better prediction methodology. As seen in chapter 4, both ATS versions still have trouble on benchmarks with high contention, scheduling very pessimistically in the case of *Delaunay*, *Intruder* and *Yada*. Still, when the system is non-overcommitted the low overhead of ATS—both the pthread and spinlock versions—allows it to be very competitive with BFGTS for benchmarks that have simple conflict patterns that also benefit from low overhead like *Genome*, *Kmeans* and *Vacation*. In the case of *Vacation*, BFGTS/Backoff can attain better performance due to its usage of "Similarity" to better detect transient conflicts. Overall we see the same trends as the non-overcommitted PTS results. BFGTS, being unable to hide serialization latency in

Benchmark	ATS-Pthread	ATS-Spinlock
Delaunay	27.1%	26.8%
Genome	1.2%	1.7%
Kmeans	1.5%	4.2%
Vacation	2.0%	5.2%
Intruder	5.1%	19.8%
Ssca2	<0.1%	<0.1%
Labyrinth	2.9%	7.9%
Yada	11.7%	10.1%

Table A.5: Contention for ATS-Pthread and ATS-Spinlock for a 16 processor LogTM system with 16 threads.

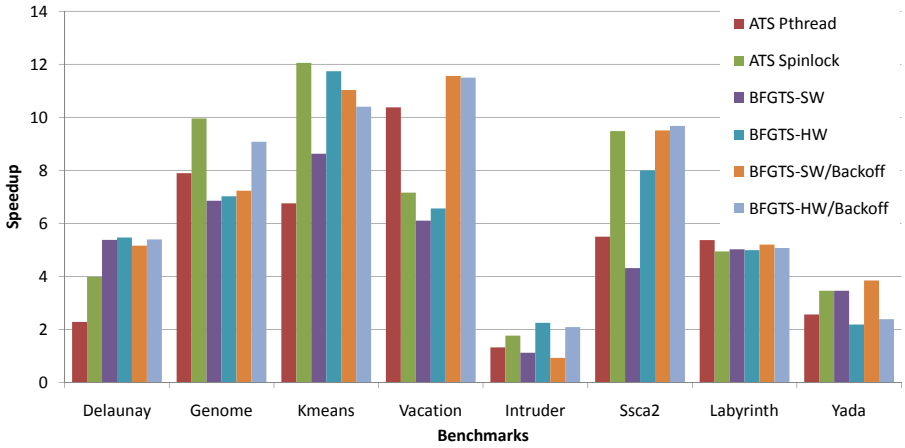


Figure A.8: Best Overall performance attained for ATS-Pthread, ATS-Spinlock, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff compared to 1 core for a 16 processor system using 16 threads.

Benchmark	BFGTS-SW	BFGTS-HW	BFGTS-SW/Backoff	BFGTS-HW/Backoff
Delaunay	23.1%	21.0%	28.0%	24.5%
Genome	1.3%	1.1%	4.7%	3.6%
Kmeans	0.9%	1.9%	5.7%	6.6%
Vacation	2.6%	2.0%	3.7%	3.2%
Intruder	31.3%	5.3%	45.1%	10.7%
Ssca2	<0.1%	<0.1%	<0.1%	<0.1%
Labyrinth	11.4%	3.9%	8.7%	4.7%
Yada	18.7%	6.7%	19.8%	7.3%

Table A.6: Contention for BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff for a 16 processor LogTM system with 16 threads.

this case sees degraded performance due to its more complicated contention management scheme when compared to the very low overhead and simple ATS-Spinlock contention manager. Comparison to ATS-Pthread is in general more favorable as it works for both an overcommitted and non-overcommitted system. The more limited number of predictions also affects the prediction accuracy of BFGTS. This can be seen in the *Yada* benchmark where the BFGTS-HW and BFGTS-HW/Backoff variants get worse performance than the BFGTS-SW and BFGTS-SW/Backoff variants. This is due to the HW variants scheduling more pessimistically because they can make a prediction faster. The SW variants have more race conditions when scheduling and can tend to schedule more optimistically. As seen in this thesis *Yada* performs best with optimistic scheduling. The more limited number

of predictions made with a non-overcommitted system leads to less profiling data using the Bloom filter operations which in turn lead to worse predictions.

Figures A.9 and A.10 show the time breakdowns of execution for the various contention managers tested. As can seen the BFGTS techniques spend more time in "Sched Begin" than the overcommitted system from chapter 4 because it is spinning to stall the transaction when a conflict is predicted. ATS still sees reduced performance for high contention benchmarks like *Delaunay* where it spends most of its time in the kernel waiting for a thread to wake up threads waiting on the condition variable, or spending time in "Sched Begin" actively spinning on a lock. It should be noted for benchmarks like *Kmeans*, where ATS-Pthread has trouble with overhead in the kernel and "Sched Begin", ATS-Spinlock eliminates most of that overhead by using much simpler spinlocks over the pthread mutexes used in ATS-Pthread and a substantial performance gain is seen.

Figures A.11, A.12, A.13 and A.14 show the sensitivity of BFGTS to the Bloom filter size for the non-overcommitted system. As with the non-overcommitted PTS results, BFGTS follows the same Bloom filter sensitivity trends as the overcommitted experiments conducted in chapter 4.

As seen in Section A.3, the main factor in the large difference in performance is the lack of being able to hide serialization latency by switching in different threads using `pthread_yield()`. These additional experiments also help expose the differences in the PTS and BFGTS algorithms and isolate the benefit gained from using "Similarity" to guide updates to confidence. By using "Similarity" BFGTS outperforms PTS by a wide margin while not requiring any application specific optimizations which is a significant result.

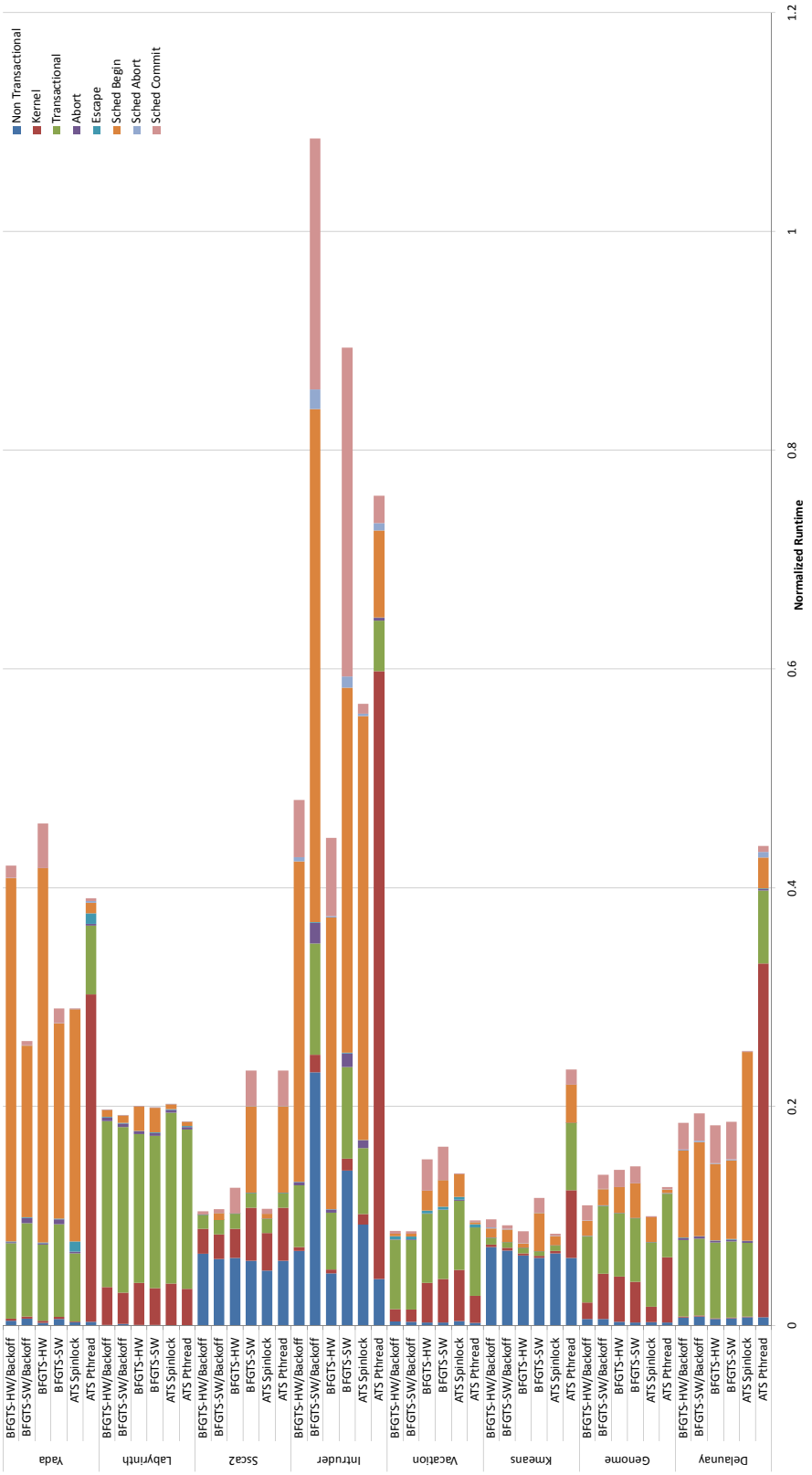


Figure A.9: Time breakdown of where ATS-Pthread, ATS-Spinlock, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff spend time executing for a 16 processor system using 16 threads normalized to a 1 core system for each benchmark.

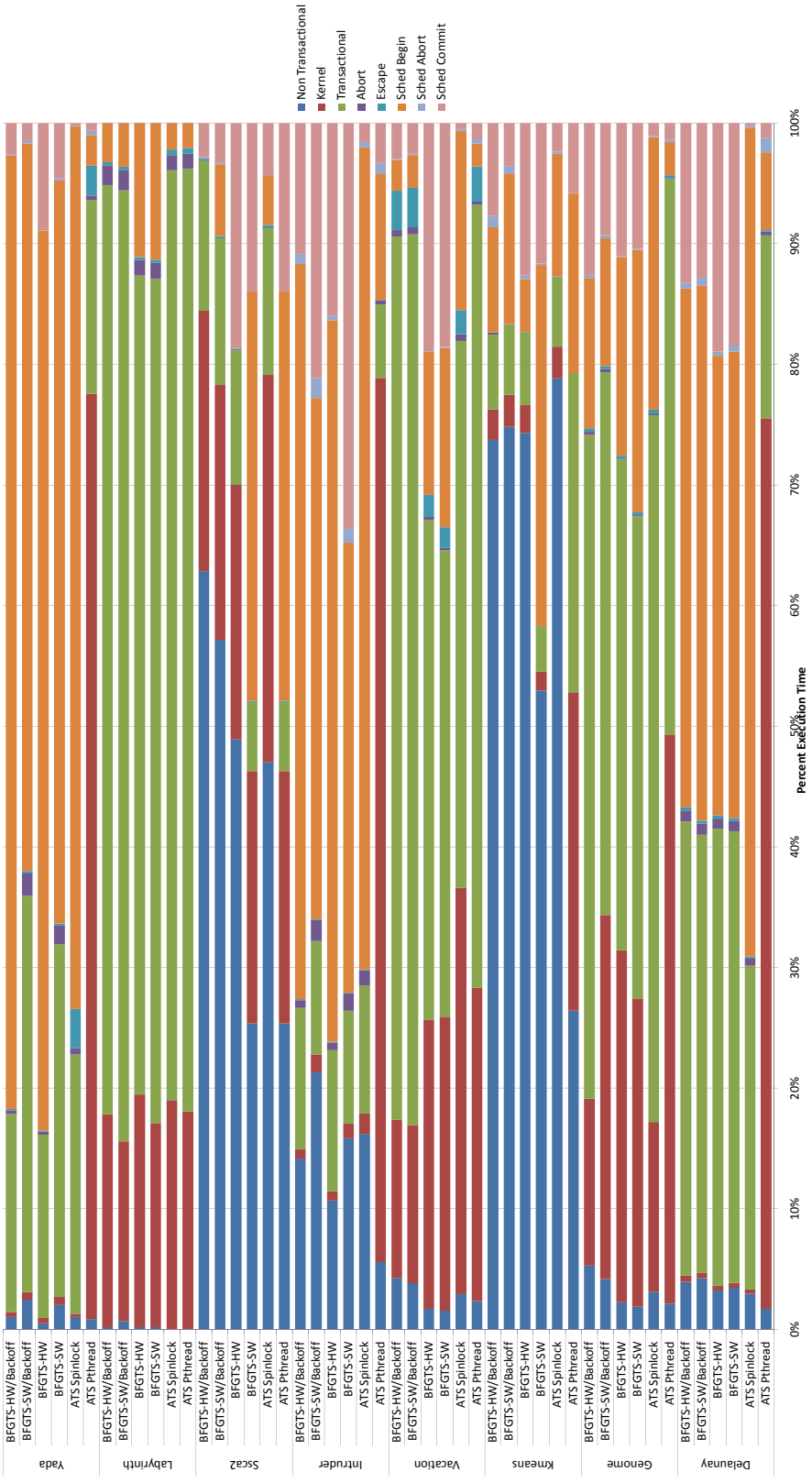


Figure A.10: Time distribution of ATS-Pthread, ATS-Spinlock, BFGTS-SW, BFGTS-HW, BFGTS-SW/Backoff and BFGTS-HW/Backoff for a 16 processor system using 16 threads

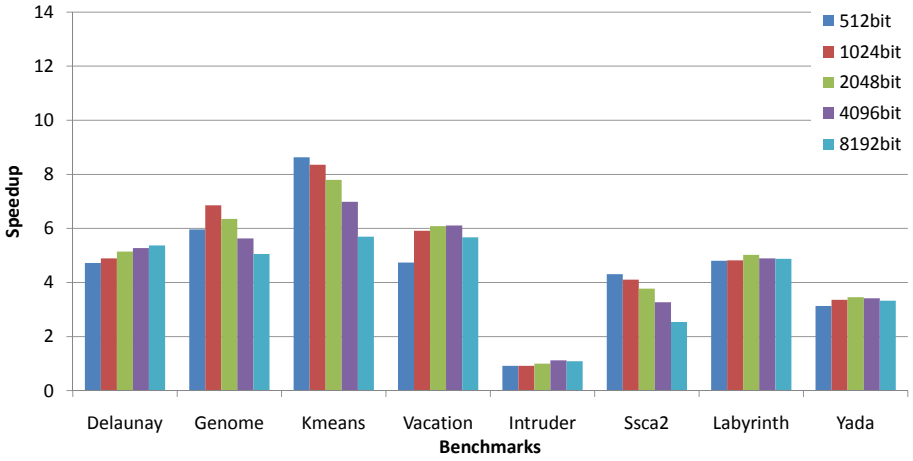


Figure A.11: Bloom filter sensitivity of BFGTS-SW for 16 processor using 16 threads.

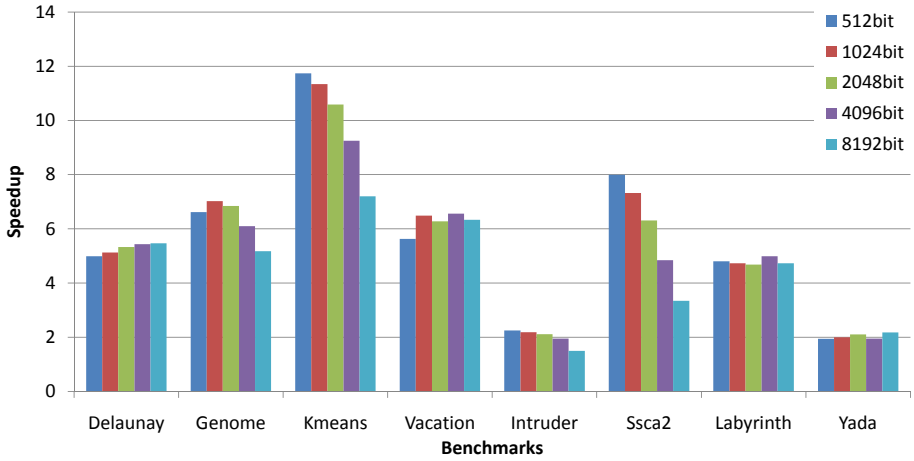


Figure A.12: Bloom filter sensitivity of BFGTS-HW for 16 processor using 16 threads.

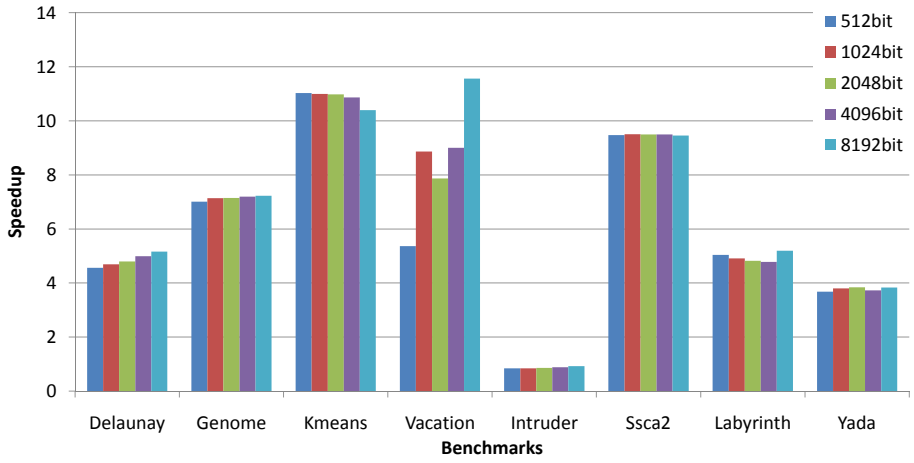


Figure A.13: Bloom filter sensitivity of BFGTS-SW/Backoff for 16 processor using 16 threads.

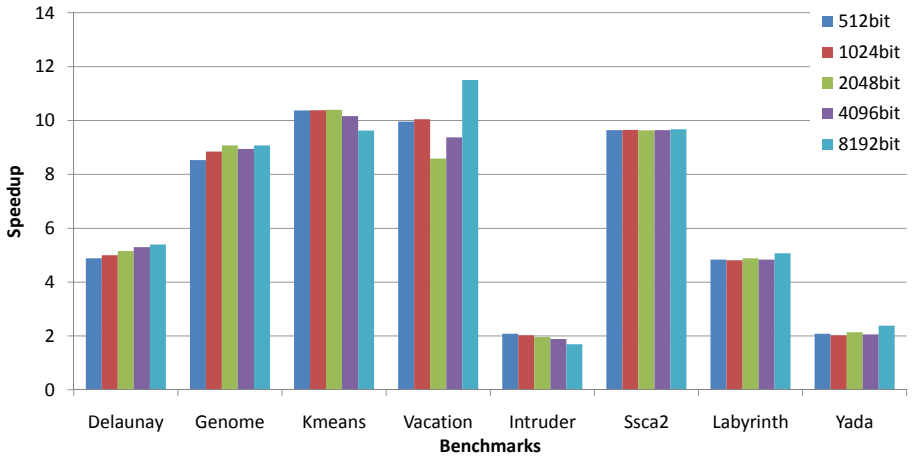


Figure A.14: Bloom filter sensitivity of BFGTS-HW/Backoff for 16 processor using 16 threads.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Power4 system microarchitecture. http://www-03.ibm.com/systems/p/-hardware/whitepapers/power4.html, 2001.

[2] AMD Announces World's First 64-Bit, x86 Multi-Core Processors For Servers And Workstations At Second-Anniversary Celebration Of AMD Opteron Processor. *AMD News Room*, 2005.

[3] Intel Has Double Vision: First Multi-Core Silicon Production Begins. *Intel Press Room*, 2005.

[4] Intel 64 and IA-32 Architectures Software Developer's Manual. *Intel Developer Manuals*, 2, Nov 2008.

[5] Tilepro64 processor. *Tilera Product Brief*, 2008.

[6] AMD Displays Llano Die: 4 x86 Cores, 480 Stream Processors. http://www.xbitlabs.com/news/cpu/display/ 20091111143547_AMD_Displays_Llano_Die_4_x86_Cores _480_Stream_Processors.html, 2009.

[7] Intel Core i7 Processor. http://www.intel.com/products/ processor/-corei7/specifications.htm, 2009.

[8] Intel Previews Intel Xeon 'Nehalem-EX' Processor. *Intel Press Room*, 2009.

[9] NVIDIAs Next Generation CUDA Compute Architecture: Fermi. *NVIDIA Development Whitepapers and Presentations*, 2009.

[10] OMAP 4: Mobile applications platform. *Texas Instruments Product Bullentin*, 2009.

[11] AMD Sets the New Standard for Price, Performance, and Power for the Datacenter. *AMD Newsroom*, 2010.

[12] Intel Spotlights New Extreme Edition Processor, Software Developer Resources at Game Conference. *Intel Press Room*, 2010.

[13] OCTEON II CN68XX Multi-Core MIPS64 Processors. *Cavium Networks Product Brief*, 2010.

[14] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 104–114, New York, NY, USA, 1990. ACM.

[15] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007. A later version appeared at PPoPP 08.

[16] A. Alamedldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *The Ninth International Symposium on High-Performance Computer Architecture*. Feb 2003.

[17] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[18] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[19] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Proc. 14th European Conference on Parallel Processing*, pages 719–728, Aug 2008. Springer-Verlag Lecture Notes in Computer Science volume 5168.

[20] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HIPEAC '09: Proc. 4th International Conference on High Performance and Embedded Architectures and Compilers*, pages 4–18, Jan 2009. Springer-Verlag Lecture Notes in Computer Science volume 5409.

[21] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 253–262, New York, NY, USA, 2010. ACM.

[22] T. Bai, X. Shen, C. Zhang, W. N. Scherer III, C. Ding, and M. L. Scott. A key-based adaptive transactional memory executor. In *Proceedings of the NSF Next Generation Software Program Workshop*. Mar 2007. Invited paper. Also available as TR 909, Department of Computer Science, University of Rochester, Dec 2006.

[23] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 506–517, Washington, DC, USA, 2005. IEEE Computer Society.

[24] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 91–100, New York, NY, USA, 2010. ACM.

[25] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

[26] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[27] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 156–167, New York, NY, USA, 2009. ACM.

[28] G. Blake, R. G. Dreslinski, and T. Mudge. Bloom filter guided transaction scheduling. In *The 17th IEEE International Symposium on High Performance Computer Architecture*, Feb 2011.

[29] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 302–313, New York, NY, USA, 2010. ACM.

[30] G. Blake and T. Mudge. Duplicating and verifying logtm with os support in the m5 simulator. *Workshop on Duplicating, Deconstructing and Debunking*, 2007.

[31] G. Blake, T. Mudge, S. Biles, N. Chong, E. Ozer, and R. G. Dreslinski. Contention management for a hardware transactional memory. USA Patent Application #20090138890, Nov. 2008.

[32] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[33] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[34] C. Blundell, A. Raghavan, and M. M. Martin. Retcon: Transactional repair without replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 258–269, New York, NY, USA, 2010. ACM.

[35] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[36] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[37] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, Sep 2008.

[38] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[39] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, New York, NY, USA, 1977. ACM.

[40] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[41] L. Ceze, P. Montesinos, C. von Praun, and J. Torrellas. Colorama: Architectural support for data-centric synchronization. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*. Feb 2007.

[42] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. Jun 2006.

[43] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *13th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2007.

[44] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 27–40, New York, NY, USA, 2010. ACM.

[45] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. *Microarchitecture, IEEE/ACM International Symposium on*, 0:39–50, 2010.

[46] C. Click. Azuls experiences with hardware transactional memory. In *In HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[47] I. Corporation. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors. *Intel White Papers*, 2008.

[48] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. pages 157–168, mar 2009.

[49] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134. Aug 2008.

[50] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *PODC '09: Proc. 28th ACM Symposium on Principles of Distributed Computing*, Aug 2009.

[51] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, Feb 2010.

[52] R. G. Dreslinski. *Something about NTC circuits and architecture. TODO Change this*. PhD thesis, The University of Michigan, Ann Arbor, 2011.

[53] C. Ellis. Concurrent search and insertion in avl trees. *Computers, IEEE Transactions on*, C-29(9):811 –817, Sept 1980.

[54] P. Enslow, Jr. Multiprocessor organization—a survey. *ACM Comput. Surv.*, 9:103–129, March 1977.

[55] S. Everman and L. Eeckhout. A memory-level parallelism aware fetch policy for smt processors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 240–249, Washington, DC, USA, 2007. IEEE Computer Society.

[56] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 362–370. ACM, Jun 2010.

[57] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2010.

[58] P. Glaskowsky. Prescott pushes pipelining limits. *Microprocessor Report*, February 2004.

[59] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[60] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical Report 2006-04-01, University of Washington Department of Computer Science & Engineering, Apr 2006.

[61] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 258–264, New York, NY, USA, Jul 2005. ACM Press.

[62] L. Gwennap. Armada 628 Sails Into Mobile Lead. *Microprocessor Report*, October 2010.

[63] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The stanford hydra cmp. *IEEE Micro*, 20(2):71–84, 2000.

[64] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[65] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[66] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522 – 529, May 2003.

[67] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[68] M. D. Hill and M. R. Marty. Amdahls law in the multicore era. *IEEE COMPUTER*, 2008.

[69] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 145–156. ACM, mar 2009.

[70] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit power efficient sparc soc (niagara2). In *ISPD '07: Proceedings of the 2007 international symposium on Physical design*, pages 2–2, New York, NY, USA, 2007. ACM.

[71] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[72] M. Kulkarni, L. P. Chew, and P. Keshav. Using transactions in delaunay mesh generation. *Workshop on Transactional Memory Workloads*, 2006.

[73] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 81–92, Dec 2003.

[74] B.-J. Kwak, N.-O. Song, and L. Miller. Performance analysis of exponential backoff. *Networking, IEEE/ACM Transactions on*, 13(2):343–355, April 2005.

[75] N. B. Lakshminarayana and H. Kim. Effect of instruction fetch and memory scheduling on gpu performance. In *Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP 2010*, 2010.

[76] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 25:1–25:12, New York, NY, USA, 2009. ACM.

[77] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 1st edition, 2006.

[78] C. Lee. An algorithm for path connections and its applications. In *IRE Transactions on Electronic Computers*, 1961.

[79] V. C. S. Lee and K.-W. Lam. Conflict free transaction scheduling using serialization graph for real-time databases. *J. Syst. Softw.*, 55(1):57–65, 2000.

[80] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[81] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, New York, NY, USA, 1977. ACM.

[82] A. McDonald, J. Chung, D. C. Brian, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. pages 53–65. Jun 2006.

[83] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski. Improving distributed join efficiency with extended bloom filter operations. *Advanced Information Networking and Applications, International Conference on*, 0:187–194, 2007.

[84] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. Feb 2006.

[85] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370. ACM Press, New York, NY, USA, Oct 2006.

[86] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 221–234, 2009.

[87] S. Pant and G. Byrd. A case for using value prediction to improve performance of transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, Feb 2009.

[88] S. Pant and G. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *CF '09: Proc. 6th ACM conference on Computing frontiers*, pages 11–20, May 2009.

[89] S. E. Raasch and S. K. Reinhardt. Applications of thread prioritization in smt processors. In *In Proceedings of the 1999 Multithreaded Execution, Architecture, and Compilation Workshop*, 1999.

[90] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[91] H. E. Ramadan, C. J. Rossbach, O. S. Hofmann, and E. Witchel. Dependence-aware transactional memory. In *The 41st Annual International Symposium on Microarchitecture*. Nov 2008.

[92] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. *SIGARCH Comput. Archit. News*, 35(2):92–103, 2007.

[93] C. Rossbach, O. Hofmann, and E. Witchel. Is transactional memory programming actually easier? In *WDDD '09: Proc. 8th Workshop on Duplicating, Deconstructing, and Debunking*, Jun 2009.

[94] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 87–102, New York, NY, USA, 2007. ACM.

[95] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133. IEEE Computer Society, 2007.

[96] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[97] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005.

[98] G. Sharma, B. Estrade, and C. Busch. Window-based greedy contention management for transactional memory. In *DISC'10: Proceedings of the 24th International Symposium on Distributed Computing*, volume 6343 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin / Heidelberg, September 2010.

[99] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[100] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. Jun 2007.

[101] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[102] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 441–450, New York, NY, USA, 2010. ACM.

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

[104] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[105] F. Tabba, A. W. Hay, and J. R. Goodman. Transactional value prediction. In *TRANSACT '09: 4th Workshop on Transactional Computing*, Feb 2009.

[106] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO '09: Proceedings of the 2009 42nd IEEE/ACM International Symposium on Microarchitecture*, 2009.

[107] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 191–202, New York, NY, USA, 1996. ACM.

[108] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[109] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[110] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.

[111] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*. Feb 2007.

[112] L. Yen, S. Draper, and M. Hill. Notary: Hardware techniques to enhance signatures. pages 234–245, Nov. 2008.

[113] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 169–178, New York, NY, USA, 2008. ACM.

[114] B. Zhai, R. G. Dreslinski, D. Blaauw, T. Mudge, and D. Sylvester. Energy efficient near-threshold chip multi-processing. In *Proceedings of the 2007 international symposium on Low power electronics and design*, ISLPED '07, pages 32–37, New York, NY, USA, 2007. ACM.

[115] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and nontransactional actions. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.